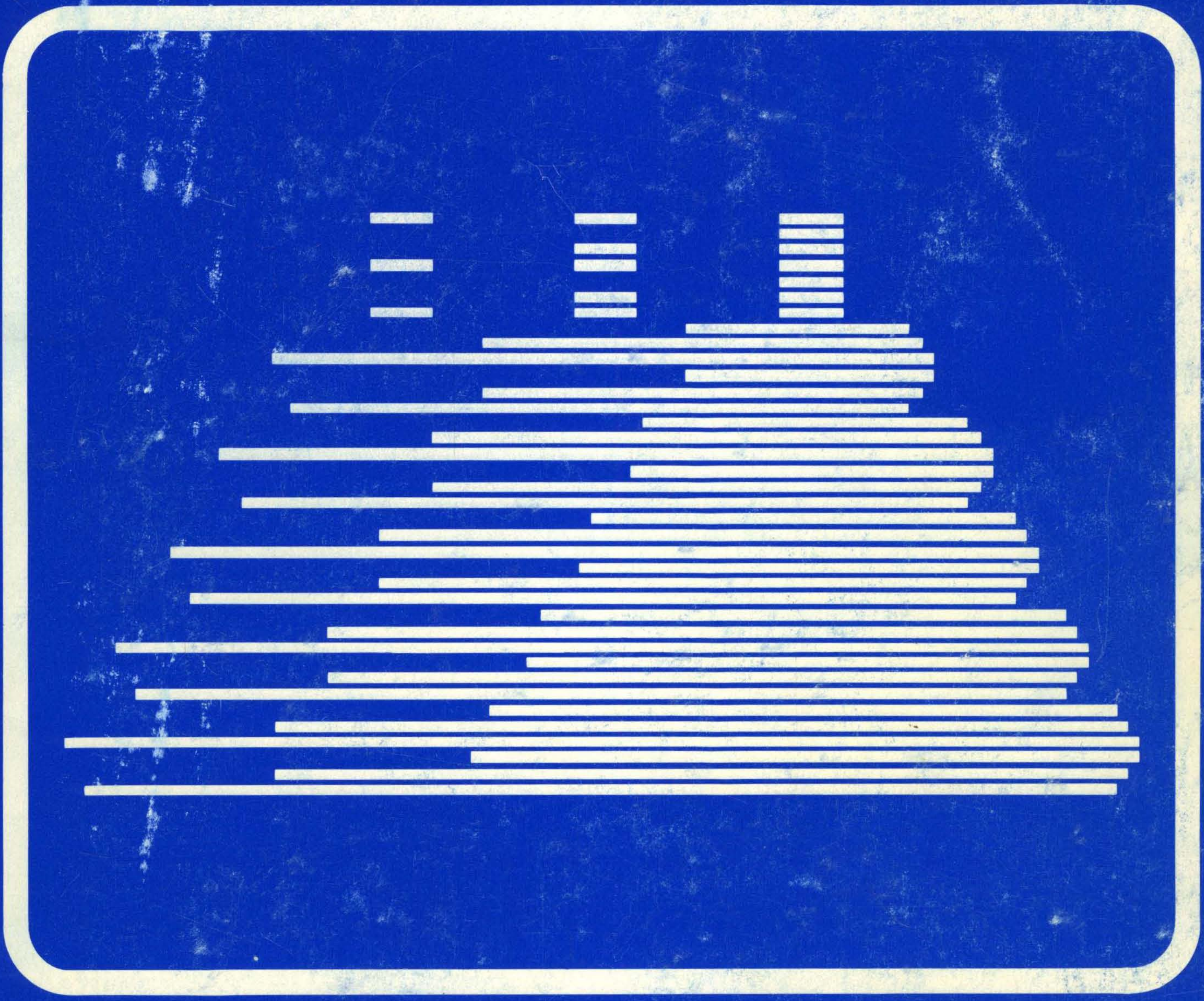
XEROX



October, 1983

Interlisp Reference Manual

Interlisp Reference Manual

October, 1983

BACKGROUND AND ACKNOWLEDGEMENTS

1 A BRIEF HISTORY OF INTERLISP

Interlisp began with an implementation of the Lisp programming language for the PDP-1 at Bolt, Beranek and Newman in 1966. It was followed in 1967 by 940 Lisp, an upward compatible implementation for the SDS-940 computer. 940 Lisp was the first Lisp system to demonstrate the feasibility of using software paging techniques and a large virtual memory in conjunction with a list-processing system [Bobrow & Murphy, 1967]. 940 Lisp was patterned after the Lisp 1.5 implementation for CTSS at MIT, with several new facilities added to take advantage of its timeshared, on-line environment. DWIM, the Do-What-I-Mean error correction facility, was introduced into this system in 1968 by Warren Teitelman [Teitelman, 1969].

The SDS-940 computer was soon outgrown, and in 1970 BBN-Lisp, an upward compatible Lisp system for the PDP-10, was implemented under the Tenex operating system. With the hardware paging and 256K of virtual memory provided by Tenex, it was practical to provide more extensive and sophisticated user support facilities, and a library of such facilities began to evolve. In 1972, the name of the system was changed to Interlisp, and its development became a joint effort of the Xerox Palo Alto Research Center and Bolt, Beranek and Newman. The next few years saw a period of rapid growth and development of the language, the system and the user support facilities, including the record package, the file package, and Masterscope. This growth was paralleled by a corresponding increase in the size and diversity of the Interlisp user community.

In 1974, an implementation of Interlisp was begun for the Xerox Alto, an experimental microprogrammed personal computer [Thacker et al., 1979]. AltoLisp [Deutsch, 1973] introduced the idea of providing a specialized, microcoded instruction set that modelled the basic operations of Lisp more closely than a general-purpose instruction set could – and as such was the first true "Lisp machine". AltoLisp also served as a departure point for Interlisp-D, the implementation of Interlisp for the Xerox 1100 Series of single-user computers, which was begun in 1979 [Sheil & Masinter, 1983].

In 1976, partially as a result of the AltoLisp effort, a specification for the Interlisp "virtual machine" was published [Moore, 1976]. This attempted to specify a small set of "primitive" operations which would support all of the higher level user facilities, which were nearly all written in Lisp. Although incomplete and written at a level which preserved too many of the details of the Tenex operating system, this document proved to be a watershed in the development of Interlisp, since it gave a clear definition of a (relatively) small kernel whose implementation would suffice to port Interlisp to a new environment. This was decisive in enabling the subsequent implementations and preserving the considerable investment that had been made in developing Interlisp's sophisticated user programming tools.

Most recently, the implementation of Interlisp on personal workstations (such as Interlisp-D) has extended Interlisp in major ways. Most striking has been the incorporation of interactive graphics and local area network facilities. Not only have these extensions expanded the range of applications for which Interlisp is being used (to include interactive interface design, network protocol experimentation and the development of specialized workstations, among others) but the personal machine capabilities have had a major impact on the Interlisp programming system itself. Whereas the original Interlisp user interface assumed a very limited (teletype) channel to the user, the use of interactive graphics and the "mouse" pointing device has radically expanded the bandwidth of communication between the user and the machine. This has enabled completely new styles of interaction with the user (e.g., the use of multiple windows to provide several different interaction channels with the user) and these have provided both new programming tools and new ways of viewing and using the existing ones. In addition, the increased use of local area networks (such as the Ethernet) has expanded the horizon of the Interlisp user beyond the local machine to a whole community of machines, processes and services. Large portions of this manual are devoted to documenting the enhanced environment that has resulted from these developments.

2 INTERLISP IMPLEMENTATIONS

Development of Interlisp-10 was, until approximately 1978, funded by the Advanced Research Projects Administration of the Department of Defence (DARPA). Subsequent developments, which have emphasized the personal workstation facilities, have been sponsored by the Xerox Corporation, with some contributions from members of the Interlisp user community.

Interlisp is currently implemented on a number of different machines. Each distinct Interlisp implementation is denoted by a suffix: Interlisp-10 is the implementation for the DEC PDP-10 family of machines running either the TENEX or TOPS-20 operating systems. Interlisp-D is the implementation for the Xerox 1100 series of machines (1100, 1108, 1132). Interlisp-VAX is the implementation for the DEC VAX family, under either the VMS or UNIX operating systems. Interlisp-Jericho is the implementation for the BBN Jericho, a internal research computer built by Bolt, Beranek and Newman. Other implementations of Interlisp have been reported (e.g. Interlisp-370, Interlisp-B5700), but are not widely used or actively maintained.

This manual is a reference manual for all Interlisp implementations. Where necessary, notes indicate when features are only available in certain implementations. For some implementations, there is also a companion "Users Guide" which documents features which are completely unique to that machine; for example, how to turn on the system, logging on, and unique facilities which link Interlisp to the host environment or operating system.

3 ACKNOWLEDGEMENTS

The Interlisp system is the work of many people – after nearly twenty years, too many even to list, much less detail their contributions. Nevertheless, some individuals cannot go unacknowledged:

Warren Teitelman, more than anyone else, made Interlisp "happen". Warren designed and implemented large parts of several generations of Interlisp, including the initial versions of most of the user facilities, coordinated the system development and assembled and edited the first four editions of the Interlisp reference manual.

Dan Bobrow was a principal designer of Interlisp's predecessors, has contributed to the implementation of several generations of Interlisp, and (in collaboration with others) made major advances in the underlying architecture, including the spaghetti stack, the transaction garbage collector, and the block compiler.

BACKGROUND AND ACKNOWLEDGEMENTS

Larry Masinter is the principal architect of the current Interlisp system, has contributed extensively to several implementations, and has designed and developed major extensions to both the Interlisp language and the programming environment.

Ron Kaplan has decisively shaped many of the programming language extensions and user facilities of Interlisp, has played a key role in two implementations and has contributed extensively to the design and content of the Interlisp reference manual.

Peter Deutsch designed the AltoLisp implementation of Interlisp which developed several key design insights on which the current generation of personal machine implementations depend.

Alice Hartley and Daryle Lewis were key contributors to implementations of Interlisp at Bolt, Beranek and Newmann.

No matter where one ends this list, one is tempted to continue. Many others who contributed to particular implementations or revisions are acknowledged in the documentation for those systems. Following that tradition, this manual, which was prepared primarily to document the extensions implemented by the Interlisp-D group at Xerox, Palo Alto, acknowledges, in addition to those listed above, the work of

Dick Burton who designed and implemented most of the interactive display facilities

Bill van Melle who designed and implemented the local area network facilities and multiple process extensions

and the contributions of Beau Sheil, Alan Bell, Steve Purcell, Steve Gadol, Joni White, Don Charnley, Willie Sue Haugeland and the many others who have helped and contributed to the development of Interlisp-D.

Like Interlisp itself, the Interlisp Reference Manual is the work of many people, some of whom are acknowledged above. This edition was designed, edited and produced by Michael Sannella of the Interlisp-D group at Xerox, Palo Alto. It is a substantial revision of the previous edition [Teitelman et al., 1978] — it has been completely reorganized, updated in most sections, and extended with a large amount of new material. In addition to material taken from the previous edition, this edition contains major extensions contributed by members of the Interlisp-D group and contributions from other Interlisp developers at the Information Sciences Institute of the University of Southern California and Bolt Beranek and Newman.

Interlisp is not designed by a formal committee. It grows and changes in response to the needs of those who use it. Contributions and discussion from the user community remain, as they have always been, warmly welcome.

4 REFERENCES

[Bobrow & Murphy, 1967]
Bobrow, D.G., and Murphy, D.L., "The Structure of a LISP System Using Two Level Storage" — *Communications of the ACM*, Vol. 10, 3, (March, 1967).

[Bobrow & Wegbreit, 1973]
Bobrow, D.G., and Wegbreit, B., "A Model and Stack Implementation for Multiple Environments" — *Communications of the ACM*, Vol. 16, 10, (October 1973).

[Deutsch, 1973] Deutsch, L.P., "A Lisp machine with very compact programs" — *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford, (1973).

[Moore, 1976] Moore, J.S., "The Interlisp Virtual Machine Specification" — Xerox PARC, CSL-76-5, (1976).

[Sheil & Masinter, 1983]
Sheil, B., and Masinter, L.M. (eds.), "Papers on Interlisp-D" — Xerox PARC, CIS-5 (Revised), (1983).

[Teitelman, 1969] Teitelman, W., "Toward a Programming Laboratory" — *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, (1969).

[Teitelman, et al., 1972]
Teitelman, W., Bobrow, D.G., Hartley, A.K. Murphy, D.L., *BBN-LISP TENEX Reference Manual* — Bolt Beranek and Newman, (July 1971, first revision February 1972, second revision August 1972).

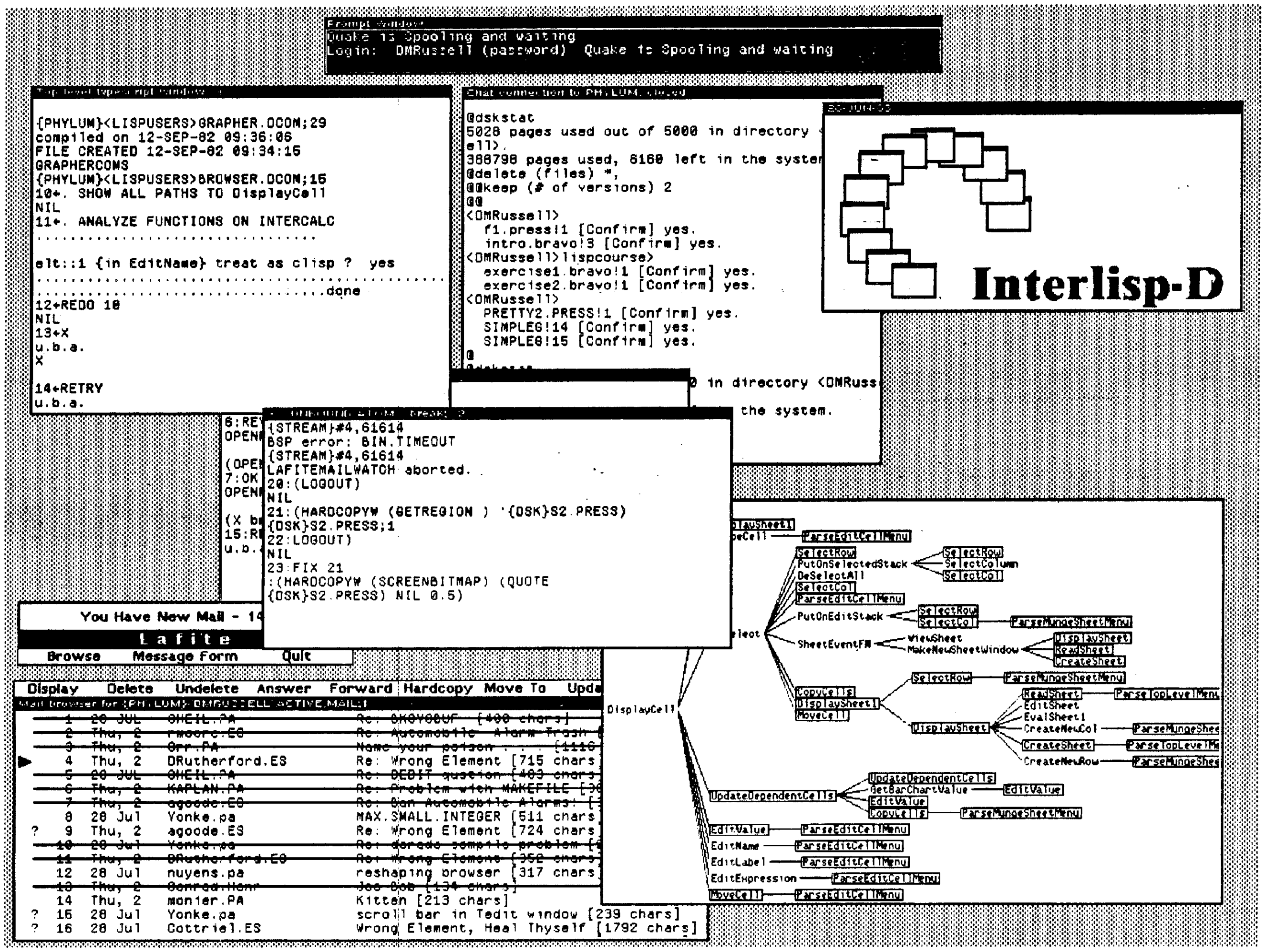
[Teitelman, et al., 1978]
Teitelman, W., et al., *The Interlisp Reference Manual* — Xerox PARC, (October 1978).

[Thacker, et al., 1979]
Thacker, C., Lampson, B., and Sproull, R., "Alto: A personal computer" — Xerox PARC,CSL-79-11, (August, 1979).

TABLE OF CONTENTS

Chapter 19 INTERLISP-D DISPLAY FACILITIES

Chapter 21 ETHERNET



```
Quake is Spooling and waiting
Login:  DMRussell (password)  Quake is Spooling and waiting
```

Top level typescript window

```
{PHYLUM}<LISPUSERS>GRAPHER.DCOM;29
compiled on 12-SEP-82 09:36:06
FILE CREATED 12-SEP-82 09:34:15
GRAPHERCOMS
{PHYLUM}<LISPUSERS>BROWSER.DCOM;15
10←. SHOW ALL PATHS TO DisplayCell
NIL
11←. ANALYZE FUNCTIONS ON INTERCALC
...............................

elt::1 {in EditName} treat as clisp ?  yes
.....................................................
.....................................done
12←REDO 10
NIL
13←X
u.b.a.
X

14←RETRY
u.b.a.
```

Chat connection to PHYLUM, closed

```
@dskstat
5028 pages used out of 5000 in directory <
ell>.
388798 pages used, 8160 left in the system
@delete (files) *,
@@keep (# of versions) 2
@@
<DMRussell>
  f1.press!1 [Confirm] yes.
  intro.bravo!3 [Confirm] yes.
<DMRussell>lispcourse>
  exercise1.bravo!1 [Confirm] yes.
  exercise2.bravo!1 [Confirm] yes.
<DMRussell>
  PRETTY2.PRESS!1 [Confirm] yes.
  SIMPLEG!14 [Confirm] yes.
  SIMPLE8!15 [Confirm] yes.
@
0 in directory <DMRuss
 the system.
```

```
6:RE
OPEN

(OPE
7:OK
OPEN

(X b
15:R
u.b.
```

```
{STREAM}#4,61614
BSP error: BIN.TIMEOUT
{STREAM}#4,61614
LAFITEMAILWATCH aborted.
20:(LOGOUT)
NIL
21:(HARDCOPYW (GETREGION ) '{DSK}S2.PRESS)
{DSK}S2.PRESS;1
22:LOGOUT)
NIL
23:FIX 21
:(HARDCOPYW (SCREENBITMAP) (QUOTE
{DSK}S2.PRESS) NIL 0.5)
```

~~1 20 JUL OHEIL.PA Re: OKOYOBUF [400 chars]~~
~~2 Thu, 2 rmoore.ES Re: Automobile Alarm Trash~~
~~3 Thu, 2 Orr.PA Name your poison . . . [1116~~
▶ 4 Thu, 2 DRutherford.ES Re: Wrong Element [715 chars
~~5 20 JUL OHEIL.PA Re: DEBIT question [403 chars~~
~~6 Thu, 2 KAPLAN.PA Re: Problem with MAKEFILE [3~~
~~7 Thu, 2 agoode.ES Re: Ban Automobile Alarms! [~~
8 28 Jul Yonke.pa MAX.SMALL.INTEGER [511 chars
? 9 Thu, 2 agoode.ES Re: Wrong Element [724 chars
~~10 28 Jul Yonke.pa Re: dorado compile problem [~~
~~11 Thu, 2 DRutherford.ES Re: Wrong Element [352 chars~~
12 28 Jul nuyens.pa reshaping browser [317 chars
~~13 Thu, 2 Conrad.Henr Joe Bob [134 chars]~~
14 Thu, 2 monier.PA Kitten [213 chars]
? 15 28 Jul Yonke.pa scroll bar in Tedit window [239 chars]
? 16 28 Jul Cottriel.ES Wrong Element, Heal Thyself [1792 chars]

CHAPTER 1

INTRODUCTION

Interlisp is a *programming system.* A programming system consists of a programming *language,* a large number of predefined programs (or *functions,* to use the Lisp terminology) that can be used either as direct user commands or as subroutines in user programs, and an *environment* that supports the programmer by providing a variety of specialized programming tools. The language and predefined functions of Interlisp are rich, but similar to those of other modern programming languages. The Interlisp programming environment, on the other hand, is very distinctive. Its most salient characteristic is an integrated set of programming tools which know enough about Interlisp programming so that they can act as semi-autonomous, intelligent "assistants" to the programmer. In addition, the environment provides a completely self-contained world for creating, debugging and maintaining Interlisp programs.

This manual describes all three components of the Interlisp system. There are discussions about the content and structure of the language, about the pieces of the system that can be incorporated into user programs, and about the environment. The line between user code and the environment is thin and changing. Most users extend the environment with some special features of their own. Because Interlisp is so easily extended, the system has grown over time to incorporate many different ideas about effective and useful ways to program. This gradual accumulation over many years has resulted in a rich and diverse system. That is the reason this manual is so large.

Whereas the rest of this manual describes the individual pieces of the Interlisp system, this chapter attempts to describe the whole system—language, environment, tools, and the otherwise unstated philosophies that tie it all together. It is intended to give a global view of Interlisp to readers approaching it for the first time.

1.1 INTERLISP AS A PROGRAMMING LANGUAGE

This manual does not contain an introduction to programming in Lisp. Sadly, primers and teaching materials for Lisp are few and quickly become dated. [Winston & Horn, 1981] discuss Lisp and its applications, but focus on MacLisp, with only a limited section on Interlisp in an appendix. [Siklossy, 1976] and [Weissman, 1967] are both sound, but a little dated. In this section, we simply highlight a few key points about Lisp on which much of the later material depends.

The Lisp family of languages (e.g., Interlisp, UCI Lisp [Meehan, 1979], FranzLisp [Foderaro, 1979], MacLisp [Moon, 1974], Lisp Machine Lisp [Weinreb & Moon, 1979], etc.) shares a common structure in which large programs (or functions) are built up by composing the results of smaller ones. Although Interlisp, like most modern Lisps, allows programming in almost any style one can imagine, the natural style of Lisp is functional and recursive, in that each function computes its result by selecting from or building upon the values given to it and then passing that result back to its caller (rather than by producing "side-effects" on external data structures, for example). A great many applications can be written in Lisp in this purely functional style, which is encouraged by the simplicity with which Lisp functions can be composed together.

Lisp is also a list-manipulation language. The essential primitive data objects of any Lisp are "atoms" (symbols or identifiers) and "lists" (sequences of atoms or lists), rather than the "characters" or "numbers" of more conventional programming languages (although these are also present in all modern Lisps). Each Lisp dialect has a set of operations that act on atoms and lists, and these operations comprise the core of the language.

Invisible in the programs, but essential to the Lisp style of programming, is an automatic memory management system (an "allocator" and a "garbage collector"). Allocation of new storage occurs automatically whenever a new data object is created. Conversely, that storage is automatically reclaimed for reuse when no other object makes reference to it. Automatic allocation and deallocation of memory is essential for rapid, large scale program development because it frees the programmer from the task of maintaining the details of memory administration, which change constantly during rapid program evolution.

A key property of Lisp is that it can represent Lisp function definitions as pieces of Lisp list data. Each subfunction "call" (or *function application*) is written as a list in which the function is written first, followed by its arguments. Thus, `(PLUS 1 2)` is a list structure representation of the expression `1 + 2`. Each program can be written as a list of such function applications. This representation of program as data allows one to apply the same operations to programs that one uses to manipulate data, which makes it very straightforward to write Lisp programs which look at and change *other Lisp programs.* This, in turn, makes it easy to develop programming tools and translators, which was essential in enabling the development of the Interlisp environment.

One result of this ability to have one program examine another is that one can extend the Lisp programming language itself. If some desired programming idiom is not supported, it can be added simply by defining a function that translates the desired expression into simpler Lisp. Interlisp provides extensive facilities for users to make this type of language extension. In addition, the CLISP (Conversational LISP) package provides definitions for several commonly used programming constructs (`if ... then ... else`, `for` and `do` loops, etc.) that make many programs easier to express. Using this ability to extend itself, Interlisp has incorporated many of the constructs that have been developed in other modern programming languages.

1.2 INTERLISP AS AN INTERACTIVE ENVIRONMENT

Interlisp programs should not be thought of as autonomous, external files of source code. All Interlisp programming takes place within the Interlisp environment, which is a completely self-sufficient environment for developing and using Interlisp programs. Not only does the environment contain the obvious programming facilities (e.g., program editors, compilers, debuggers, etc.), but it also contains a variety of tools which assist the user by "keeping track" of what happens, so the user doesn't have to. For example, the Interlisp file package notices when programs or data have been changed, so that the system will know what needs to be saved at the end of the session. The "residential" style, where one stays within the environment throughout the development, from initial program definition through final debugging, is essential for these tools to operate. Furthermore, this same environment is available to support the final production version, some parts providing run time support and other parts ignored until the need arises for further debugging or development.

For terminal interaction with the user, Interlisp provides a "Read-Eval-Print" loop. That is, whatever the user types in is `READ` by the system, executed (or "`EVAL`"-uated) and the result is `PRINT`-ed onto the terminal. (This interaction is also recorded by the programmer's assistant, described below, so the user can ask to do an action again, or even to undo the effects of a previous action.) Although each interactive terminal listener (or "executive") defines a few specialized commands, most of the interaction will consist of simple evaluations of ordinary Lisp expressions. Thus, instead of specialized terminal commands for operations like manipulating the user's files, actions like this are carried out simply by typing the same expressions that one would use to accomplish them inside a Lisp program. This creates a very rich, simple and uniform set of interactive commands, since any Lisp expression can be typed at a command executive and evaluated immediately.

In normal use, one writes a program (or rather, "defines a function") simply by typing in an expression that invokes the "function defining" function (DEFINEQ), giving it the name of the function being defined and its new definition. The newly defined function can be executed immediately, simply by using it in a Lisp expression. Although most Interlisp code is normally run compiled (for reasons of efficiency), the initial versions of most programs, and all of the user's terminal interactions, will be run interpreted. Eventually, as a function gets larger or is used in many places, it becomes more effective to compile it. Usually, by that stage, the function has been stored on a file and the whole file (which may contain many functions) is compiled at once. DEFINEQ, the compiler (COMPILE), and the interpreter (EVAL), are all themselves Lisp functions that use the ability to treat other Lisp expressions and programs as data.

In addition to these basic programming tools, Interlisp also provides a wide variety of programming support mechanisms:

Structure editor
: Since Interlisp programs are represented as list structure, Interlisp provides an editor which allows one to change the list structure of a function's definition directly.

Pretty-printer
: The pretty printer is a function that prints Lisp function definitions so that their syntactic structure is displayed by the indentation and fonts used.

Break Package
: When errors occur, the break package is called, allowing the user to examine and modify the context at the point of the error. Often, this enables execution to continue without starting over from the beginning. Within a break, the full power of Interlisp is available to the user. Thus, the broken function can be edited, data structures can be inspected and changed, other computations carried out, and so on. All of this occurs in the context of the suspended computation, which will remain available to be resumed.

DWIM
: The "Do What I Mean" package automatically fixes the user's misspellings and errors in typing.

Programmer's Assistant
: Interlisp keeps track of the user's actions during a session and allows each one to be replayed, undone, or altered.

Masterscope
: Masterscope is a program analysis and management tool which can analyze users' functions and build (and automatically maintain) a data base of the results. This allows the user to ask questions like "WHO CALLS ARCTAN" or "WHO USES COEF1 FREELY" or to request systematic changes like "EDIT WHERE ANY *(function)* FETCHES ANY FIELD OF *(the data structure)* FOO".

Record/Datatype Package
: Interlisp allows a programmer to define new data structures. This enables one to separate the issues of data access from the details of how the data is actually stored.

File Package — Files in Interlisp are managed by the system, removing the problem of ensuring timely file updates from the user. The file package can be modified and extended to accomodate new types of data.

Performance Analysis — These tools allow statistics on program operation to be collected and analyzed.

These facilities are tightly integrated, so they know about and use each other, just as they can be used by user programs. For example, Masterscope uses the structural editor to make systematic changes. By combining the program analysis features of Masterscope with the features of the structural editor, large scale system changes can be made with a single command. For example, when the lowest-level interface of the Interlisp-D I/O system was changed to a new format, the entire edit was made by a single call to Masterscope of the form `EDIT WHERE ANY CALLS '(BIN BOUT ···)`. [Burton et al., 1980] This caused Masterscope to invoke the editor at each point in the system where any of the functions in the list `'(BIN BOUT ···)` were called. This ensured that no functions used in input or output were overlooked during the modification.

The new, personal machine implementations of Interlisp, such as Interlisp-D, also provide some new user facilities, and some new, interactive graphic interfaces to some of the older Interlisp programming tools:

Multiple Processes — The multiple and independent processes allowed in Interlisp-D simplify problems which require logically separate pieces of code to operate in parrallel.

Windows — The ability to have multiple, independent windows on the display allows many different processes or activities to be active on the screen at once.

Inspector — The inspector is a display tool for examining complex data structures encountered during debugging.

The figure found at the beginning of this chapter shows a standard user display within Interlisp-D. One window displays a list of messages available for browsing, using an experimental mail reading system. This operates in parallel with the user's other activities, continually monitoring the remote mail server and watching for any new messages. The "DEdit" window is editing an Interlisp function. The "Chat" window offers a direct connection to a remote machine (this one is a remote file server). There are two nested break windows showing the environment of an interrupted evaluation. And in the lower right, there is a Masterscope display showing all the possible execution paths to some function.

Some of the newer implementations of Interlisp have embedded within them an entire operating system written in Interlisp. For the most part, that is of no concern to the user (although it is nice to know that one *can* write programs of this complexity and performance within Interlisp!). However, some of the facilities provided by this low level code allow the use of Interlisp for applications that would previously have been forced into a relatively impoverished system programming environment. In particular, Interlisp-D provides complete facilities for experimenting with distributed machines and services on a local area network, plus access to all the services that such networks provide (e.g., mail, printing, filing, etc.).

1.3 INTERLISP PHILOSOPHY

The extensive environmental support that the Interlisp system provides has developed over the years in order to support a particular style of programming called "exploratory programming" [Sheil, 1983].

INTRODUCTION

For many complex programming problems, the task of program creation is *not* simply one of writing a program to fulfill pre-identified specifications. Instead, it is a matter of exploring the problem (trying out various solutions expressed as partial programs) until one finds a good solution (or sometimes, any solution at all!). Such programs are by their very nature evolutionary; they are transformed over time from one realization into another in response to a growing understanding of the problem. This point of view has lead to an emphasis on having the tools available to analyze, alter, and test programs easily. One important aspect of this is that the tools be designed to work together in an integrated fashion, so that knowledge about the user's programs, once gained, is available throughout the environment.

The development of programming tools to support exploratory programming is itself an exploration. Noone knows all the tools that will eventually be found useful, and not all programmers want all of the tools to behave the same way. In response to this diversity, Interlisp has been shaped, by its implementors and by its users, to be easily extensible in several different ways. First, there are many places in the system where its behavior can be adjusted by the user. One way that this can be done is by changing the value of various "flags" or variables whose values are examined by system code to enable or suppress certain behavior. The other is where the user can provide functions or other behavioral specifications of what is to happen in certain contexts. For example, the format used for each type of list structure when it is printed by the pretty-printer is determined by specifications that are found on the list PRETTYPRINTMACROS. Thus, this format can be changed for a given type simply by putting a printing specification for it on that list.

Another way in which users can effect Interlisp's behavior is by redefining or changing system functions. The "Advise" capability, for instance, permits the user to modify the operation of virtually any function in the system by wrapping user code "around" the selected function. (This same philosophy extends to the break package and tracing, so almost any function in the system can be broken or traced.) Experimentation is thus encouraged and actively facilitated, which allows the user to find useful pieces of the Interlisp system which can be configured to assist with application development. This is even easier in systems like Interlisp-D, where the entire system is implemented in Interlisp, since there are extremely few places where the system's behavior depends on anything outside of Interlisp (such as a low level system implementation language).

While these techniques provide a fair amount of tailorability, the price paid is that Interlisp presents an overall appearance of complexity. There are many flags, parameters and controls that affect the behavior one sees. Because of this complexity, Interlisp tends to be more comfortable for experts, rather than casual users. Beginning users of Interlisp should depend on the default settings of parameters until they learn what dimensions of flexibility are available. At that point, they can begin to "tune" the system to their preferences.

The various implementations of Interlisp share not only this general philosophy, but a philosophy about each other also. Interlisp is available in highly compatible versions across several machines. The community of Interlisp implementors is committed to maintain this level of compatibility. One testimony to this is the existence of pieces of very old code in modern versions of Interlisp that have been inherited from the original BBN-Lisp system nearly 15 years ago. Many of the function definitions in the core of the system have not changed since 1977, over many different versions of Interlisp.

Appropriately enough, even Interlisp's underlying philosophy was itself discovered during Interlisp's development, rather than laid out beforehand. The Interlisp environment and its interactive style were first analyzed in Sandewall's excellent paper [Sandewall, 1978]. The notion of "exploratory programming" and the genesis of the Interlisp programming tools in terms of the characteristic demands of this style of programming was developed in [Sheil, 1983]. The evolution and structure of the Interlisp programming environment are discussed in greater depth in [Teitelman & Masinter, 1981].

1.4 HOW TO USE THIS MANUAL

This document is a reference manual, not a primer. We have tried to provide a manual that is complete, and that allows Interlisp users to find particular items as easily as possible. Sometimes, these goals have been achieved at the expense of simplicity. For example, many functions have a number of arguments that are rarely used. In the interest of providing a complete reference, these arguments are fully explained, even though they would normally be defaulted. There is a lot of information in this manual that is only of interest to experts.

Users should not try to read straight through this manual, like a novel. In general, the chapters are organized with overview explanations and the most useful functions at the beginning of the chapter, and implementation details towards the end. If you are interested in becoming acquainted with Interlisp using this manual, the best way would be to skim through the whole book, reading the beginning of each chapter.

A few notes about the notational conventions used in this manual:

Lisp object notation: All Interlisp objects in this manual are printed in the same font: Functions (AND, PLUS, DEFINEQ, LOAD); Variables (MAX.INTEGER, FILELST, DFNFLG); and arbitrary Interlisp expressions: (PLUS 2 3), (PROG ((A 1)) ···), etc.

Case is significant: An important piece of information, often missed by newcomers to Interlisp, is that *upper and lower case is significant.* The variable FOO is not the same as the variable foo, which is not the same as the variable Foo. By convention, most Interlisp system functions and variables are all-uppercase, but users are free to use upper and lower case for their own functions and variables as they wish.[1]

This manual contains a large number of descriptions of functions, variables, commands, etc, which are printed in the following standard format:

(FOO *BAR* *BAZ* —) [Function]

This is a description for the function named FOO. FOO has two arguments, *BAR* and *BAZ*. Some system functions have extra optional arguments that are not documented and should not be used. These extra arguments are indicated by "—".

The descriptor [Function] indicates that this is a function, rather than a [Variable], [Prog. Asst. Command], etc.. For function definitions only, this can also indicate the function "type": [NLambda Function], [NoSpread Function], or [NLambda NoSpread Function], which describes whether the function takes a fixed or variable number of arguments, and whether the arguments are evaluated or not.

[1] One exception to the case-significance rule is provided by the Interlisp CLISP facility, which allows iterative statement operators and record operations to be typed in either all-uppercase or all-lowercase letters: (for X from 1 to 5 ···) is the same as (FOR X FROM 1 TO 5 ···). The few situations where this is the case are explicitly mentioned in the manual. Generally, one should assume that case is significant.

1.5 REFERENCES

[Burton, et al., 1980] Burton, R. R., L. M. Masinter, A. Bell, D. G. Bobrow, W. S. Haugeland, R.M. Kaplan and B.A. Sheil, "Interlisp-D: Overview and Status" — in [Sheil & Masinter, 1983].

[Foderaro, 1979] Foderaro, John K., *The FRANZ LISP Manual* — University of California, Bekeley, California (1979).

[Meehan, 1979] Meehan, J. R., *The New UCI Lisp Manual* — Lawrence Erlbaum Associates, Hillsdale, New Jersey (1979).

[Moon, 1974] Moon, David, *MACLISP Reference Manual* — Version 0, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, (1974)

[Sandewall, 1978] Sandewall, Erik, "Programming in the Interactive Environmnet: The LISP Experience" — *ACM Computing Surveys*, vol 10, no 1, pp 35-72, (March 1978).

[Sheil, 1983] Sheil, B.A., "Environments for Exploratory Programming" — *Datamation*, (February, 1983) — also in [Sheil & Masinter, 1983].

[Sheil & Masinter, 1983]
Sheil, B.A. and L. M. Masinter, "Papers on Interlisp-D", Xerox PARC Technical Report CIS-5 (Revised), (January, 1983).

[Siklossy, 1976] Siklossy, L., *Let's Talk Lisp* — Prentice-Hall, Englewood Cliffs, New Jersey (1976).

[Teitelman & Masinter, 1981]
Teitelman, W. and L. M. Masinter, "The Interlisp Programming Environment" — *Computer*, vol 14, no 4, pp 25-34, (April 1981) — also in [Sheil & Masinter, 1983].

[Weinreb & Moon, 1979]
Weinreb, D. and D. Moon, *Lisp Machine Manual* — Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, (January 1979).

[Weissman, 1967] Weissman, C., *LISP 1.5 Primer* — Dickenson Publishing Company, Belmont, California (1967).

[Winston & Horn, 1981]
Winston, P. H., and B.K.P. Horn, *LISP* — Addison-Wesley, Reading, Massachusetts (1981).

References

CHAPTER 2

DATA TYPES

Interlisp is a system for the manipulation of various kinds of data; it provides a large set of built-in data types, which may be used to represent a variety of abstract objects, and the user can also define new data types which can be used exactly like built-in data types.

Each data type in Interlisp has an associated "type name," a litatom.[1] Some of the type names of built-in data types are: LITATOM, LISTP, STRINGP, ARRAYP, STACKP, SMALLP, FIXP, and FLOATP. For user data types (page 3.14), the type name is specified when the data type is created.

(DATATYPES —) [Function]

Returns a list of all type names currently defined.

(TYPENAME *DATUM*) [Function]

Returns the type name for the data type of *DATUM*.

(TYPENAMEP *DATUM TYPENAME*) [Function]

Returns T if *DATUM* is an object with type name equal to *TYPENAME*, otherwise NIL.

Note: TYPENAME and TYPENAMEP distinguish the logical data types ARRAYP, CCODEP and HARRAYP, even though they may be implemented as ARRAYPs in some Interlisp implementations.

2.1 DATA TYPE PREDICATES

Interlisp provides seperate functions for testing whether objects are of certain commonly-used types:

(LITATOM *x*) [Function]

Returns T if *x* is a litatom, NIL otherwise. Note that a number is not a litatom.

(SMALLP *x*) [Function]

Returns *x* if *x* is a small integer; NIL otherwise. (Note that the range of small integers is implementation-dependent. See page 2.36.)

(FIXP *x*) [Function]

Returns *x* if *x* is a small or large integer (between MIN.FIXP and MAX.FIXP); NIL otherwise.

(FLOATP *x*) [Function]

Returns *x* if *x* is a floating point number; NIL otherwise.

[1] In Interlisp-10, each data type also has an associated "type number." See page 22.2.

(NUMBERP *x*) [Function]

Returns *x* if *x* is a number of any type (FIXP or FLOATP), NIL otherwise.

(ATOM *x*) [Function]

Returns T if *x* is an atom (i.e. a litatom or a number); NIL otherwise.

Warning: (ATOM *x*) is NIL if *x* is an array, string, etc. In many dialects of Lisp, the function ATOM is defined equivalent to the Interlisp function NLISTP.

(LISTP *x*) [Function]

Returns *x* if *x* is a list cell, e.g., something created by CONS; NIL otherwise.

(NLISTP *x*) [Function]

(NOT (LISTP X)). Returns T if *x* is not a list cell, NIL otherwise.

(STRINGP *x*) [Function]

Returns *x* if *x* is a string, NIL otherwise.

(ARRAYP *x*) [Function]

Returns *x* if *x* is an array, NIL otherwise.

Note: In some implementations of Interlisp, ARRAYP may also return *x* if it is of type CCODEP or HARRAYP.

(HARRAYP *x*) [Function]

Returns *x* if *x* is a hash array, NIL otherwise.

Note: The empty list, () or NIL, is considered to be a litatom, rather than a list. Therefore, (LITATOM NIL) = (ATOM NIL) = T and (LISTP NIL) = NIL. Care should be taken when using these functions if the object may be the empty list NIL.

2.2 DATA TYPE EQUALITY

A common operation when dealing with data objects is to test whether two objects are equal. In some cases, such as when comparing two small integers, equality can be easily determined. However, sometimes there is more than one type of equality. For instance, given two lists, one can ask whether they are exactly the same object, or whether they are two distinct lists which contain the same elements. Confusion between these two types of equality is often the source of program errors. Interlisp supplies an extensive set of functions for testing equality:

(EQ *x* *y*) [Function]

Returns T if *x* and *y* are identical pointers; NIL otherwise. EQ should not be used to compare two numbers, unless they are small integers; use EQP instead.

(NEQ *x* *y*) [Function]

(NOT (EQ *x* *y*))

(NULL *x*) [Function]
(NOT *x*) [Function]

(EQ *x* NIL)

(EQP *x* *y*) [Function]

Returns T if *x* and *y* are EQ, or if *x* and *y* are numbers and are equal in value; NIL otherwise. For more discussion of EQP and other number functions, see page 2.36.

Note: EQP also can be used to compare stack pointers (page 7.3) and compiled code (page 5.8).

(EQUAL *x* *y*) [Function]

EQUAL returns T if *x* and *y* are (1) EQ; or (2) EQP, i.e., numbers with equal value; or (3) STREQUAL, i.e., strings containing the same sequence of characters; or (4) lists and CAR of *x* is EQUAL to CAR of *y*, and CDR of *x* is EQUAL to CDR of *y*. EQUAL returns NIL otherwise. Note that EQUAL can be significantly slower than EQ.

A loose description of EQUAL might be to say that *x* and *y* are EQUAL if they print out the same way.

(EQUALALL *x* *y*) [Function]

Like EQUAL, except it descends into the contents of arrays, hash arrays, user data types, etc. Two non-EQ arrays may be EQUALALL if their respective componants are EQUALALL.

2.3 "FAST" AND "DESTRUCTIVE" FUNCTIONS

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. The user should be aware of what these functions do, and when they should be used.

"Fast" functions: By convention, a function named by prefixing an existing function name with F indicates that the new function is a "fast" version of the old. These usually have the same definitions as the slower versions, but they compile open and run without any "safety" error checks. For example, FNTH runs faster than NTH, however, it does not make as many checks (for lists ending with anything but NIL, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, the user should only use "fast" functions in code that has already been completely debugged, to speed it up.

"Destructive" functions: By convention, a function named by prefixing an existing function with D indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, REMOVE returns a copy of a list with a particular element removed, but DREMOVE actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of APPEND is NCONC.) The user should be careful when using destructive functions that they do not inadvertantly change data structures.

2.4 LITATOMS

A "litatom" (for "literal atom") is an object which conceptually consists of a print name, a value, a function definition, and a property list. In some Lisp dialects, litatoms are also known as "symbols."

A litatom is read as any string of non-delimiting characters that cannot be interpreted as a number. The syntatic characters that delimit litatoms are called separator or break characters (see page 6.32) and normally are space, end-of-line, line-feed, ((left paren),) (right paren), " (double quote), [(left bracket), and] (right bracket). However, any character may be included in a litatom by preceding it with the escape character %. Here are some examples of litatoms:

```
A  wxyz  23SKIDDOO  %]  3.1415+17

Long% Litatom% With% Embedded% Spaces
```

Litatoms are printed by PRINT and PRIN2 as a sequence of characters with %'s inserted before all delimiting characters (so that the litatom will read back in properly). Litatoms are printed by PRIN1 as a sequence of characters without these extra %'s. For example, the litatom consisting of the five characters A, B, C, (, and D will be printed as ABC%(D by PRINT and ABC(D by PRIN1.

Litatoms can also be constructed by PACK, PACK*, SUBATOM, MKATOM, and GENSYM (which uses MKATOM).

Litatoms are unique. In other words, if two litatoms print the same, they will *always* be EQ. Note that this is *not* true for strings, large integers, floating point numbers, and lists; they all can print the same without being EQ. Thus if PACK or MKATOM is given a list of characters corresponding to a litatom that already exists, they return a pointer to that litatom, and do *not* make a new litatom. Similarly, if the read program is given as input a sequence of characters for which a litatom already exists, it returns a pointer to that litatom. Note: Interlisp is different from other Lisp dialects which allow "uninterned" litatoms.

Note: Litatoms are limited to 255 characters in Interlisp-D; 127 characters in Interlisp-10. Attempting to create a larger litatom either via PACK or by typing one in (or reading from a file) will cause an error, ATOM TOO LONG.

2.4.1 Using Litatoms as Variables

Litatoms are commonly used as variables. Each litatom has a "top level" variable binding, which can be an arbitrary Interlisp object. Litatoms may also be given special variable bindings within PROGs or function calls, which only exist for the duration of the function. When a litatom is evaluated, the "current" variable binding is returned. This is the most recent special variable binding, or the top level binding if the litatom has not been rebound. SETQ is used to change the current binding. For more information on variable bindings in Interlisp, see page 7.1.

Note: The compiler (page 12.1) treats variables somewhat differently than the interpreter, and the user has to be aware of these differences when writing functions that will be compiled. For example, variable references in compiled code are not checked for NOBIND, so compiled code will not generate unbound atom errors. In general, it is better to debug interpreted code, before compiling it for speed. The compiler offers some facilities to increase the efficiency of variable use in compiled functions. Global variables (page 12.3) can be defined so that the entire stack is not searched at each variable reference. Local variables (page 12.4) allow compiled functions to access variable bindings which are not on the stack, which reduces variable conflicts, and also makes variable lookup faster.

By convention, a litatom whose top level binding is to the litatom NOBIND is considered to have no top level binding. If a litatom has no local variable bindings, and its top level value is NOBIND, attempting to evaluate it will cause an unbound atom error.

The two litatoms T and NIL always evaluate to themselves. Attempting to change the binding of T or NIL with the functions below will generate the error ATTEMPT TO SET T or ATTEMPT TO SET NIL.

The following functions (except BOUNDP) will also generate the error ARG NOT LITATOM, if not given a litatom.

(BOUNDP *VAR*) [Function]

Returns T if *VAR* has a special variable binding (even if bound to NOBIND), or if *VAR* has a top level value other than NOBIND; otherwise NIL. In other words, if *X* is a litatom, (EVAL *X*) will cause an UNBOUND ATOM error if and only if (BOUNDP *X*) returns NIL.

(SET *VAR* *VALUE*) [Function]

Sets the "current" variable binding of *VAR* to *VALUE*, and returns *VALUE*.

Note that SET is a normal lambda spread function, so both *VAR* and *VALUE* are evaluated before it is called. Thus, if the value of X is B, and the value of Y is C, then (SET X Y) would result in B being set to C, and C being returned as the value of SET.

(SETQ *VAR* *VALUE*) [NLambda NoSpread Function]

Nlambda version of SET; *VAR* is not evaluated, *VALUE* is.[2] Thus if the value of X is B and the value of Y is C, (SETQ X Y) would result in X (not B) being set to C, and C being returned.

(SETQQ *VAR* *VALUE*) [NLambda Function]

Like SETQ except that neither argument is evaluated, e.g., (SETQQ X (A B C)) sets X to (A B C).

(GETTOPVAL *VAR*) [Function]

Returns the top level value of *VAR* (even if NOBIND), regardless of any intervening local bindings.

(SETTOPVAL *VAR* *VALUE*) [Function]

Sets the top level value of *VAR* to *VALUE*, regardless of any intervening bindings, and returns *VALUE*.

A major difference between various Interlisp implementations is the way that variable bindings are implemented. Interlisp-10 and Interlisp-Jerico use what is called "shallow" binding. Interlisp-D and Interlisp-VAX use what is called "deep" binding.

[2] Since SETQ is an nlambda, *neither* argument is evaluated during the calling process. However, SETQ itself calls EVAL on its second argument. Note that as a result, typing (SETQ VAR FORM) and SETQ(VAR FORM) to the Interlisp executive is equivalent: in both cases VAR is not evaluated, and FORM is.

In a deep binding system, a variable is bound by saving on the stack the variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding. If the variable is not found on the stack, the top level binding is retrieved from a "value cell" associated with the variable.

In a "shallow" binding system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's value cell. When a variable is accessed, its value is always found in its value cell.

GETTOPVAL and SETTOPVAL are less efficient in a shallow binding system, because they have to search the stack for rebindings; it is more economical to simply rebind variables. In a deep binding system, GETTOPVAL and SETTOPVAL are very efficient since they do not have to search the stack, but can simply access the value cell directly.

GETATOMVAL and SETATOMVAL can be used to access a variable's value cell, in either a shallow or deep binding system.

(GETATOMVAL *VAR*) [Function]

Returns the value in the value cell of *VAR*. In a shallow binding system, this is the same as (EVAL *ATM*), or simply *VAR*. In a deep binding system, this is the same as (GETTOPVAL *VAR*).

(SETATOMVAL *ATM* *VALUE*) [Function]

Sets the value cell of *VAR* to *VALUE*. In a shallow binding system, this is the same as SET; in a deep binding system, this is the same as SETTOPVAL.

2.4.2 Function Definition Cells

Each litatom has a function definition cell, which is accessed when a litatom is used as a function. The mechanism for accessing and setting the function definition cell of a litatom is described on page 5.8.

2.4.3 Property Lists

Each litatom has a property list, which allows a set of named objects to be associated with the litatom. A property list associates a name, known as a "property name" or "property", with an abitrary object, the "property value" or simply "value". Sometimes the phrase "to store on the property *x*" is used, meaning to place the indicated information on a property list under the property name *x*.

Property names are usually litatoms or numbers, although no checks are made. However, the standard property list functions all use EQ to search for property names, so they may not work with non-atomic property names. Note that the same object can be used as both a property name and a property value.

Note: Many litatoms in the system already have property lists, with properties used by the compiler, the break package, DWIM, etc. Be careful not to clobber such system properties. The variable SYSPROPS is a list of property names used by the system.

The functions below are used to manipulate the propert lists of litatoms. Except when indicated, they generate the error ARG NOT LITATOM, if given an object that is not a litatom.

(GETPROP *ATM PROP*) [Function]

Returns the property value for *PROP* from the property list of *ATM*. Returns NIL if *ATM* is not a litatom, or *PROP* is not found. Note that GETPROP also returns NIL if there is an occurrence of *PROP* but the corresponding property value is NIL; this can be a source of program errors.

Note: GETPROP used to be called GETP.

(PUTPROP *ATM PROP VAL*) [Function]

Puts the property *PROP* with value *VAL* on the property list of *ATM*. *VAL* replaces any previous value for the property *PROP* on this property list. Returns *VAL*.

(ADDPROP *ATM PROP NEW FLG*) [Function]

Adds the value *NEW* to the list which is the value of property *PROP* on the property list of *ATM*. If *FLG* is T, *NEW* is CONSed onto the front of the property value of *PROP*, otherwise it is NCONCed on the end (using NCONC1). If *ATM* does not have a property *PROP*, or the value is not a list, then the effect is the same as (PUTPROP *ATM PROP* (LIST *NEW*)). ADDPROP returns the (new) property value. Example:

```
← (PUTPROP 'POCKET 'CONTENTS NIL)
NIL
← (ADDPROP 'POCKET 'CONTENTS 'COMB)
(COMB)
← (ADDPROP 'POCKET 'CONTENTS 'WALLET)
(COMB WALLET)
```

(REMPROP *ATM PROP*) [Function]

Removes all occurrences of the property *PROP* (and its value) from the property list of *ATM*. Returns *PROP* if any were found, otherwise NIL.

(REMPROPLIST *ATM PROPS*) [Function]

Removes all occurrences of all properties on the list *PROPS* (and their corresponding property values) from the property list of *ATM*. Returns NIL.

(CHANGEPROP *X PROP1 PROP2*) [Function]

Changes the property name of property *PROP1* to *PROP2* on the property list of *X*, (but does not affect the value of the property). Returns *X*, unless *PROP1* is not found, in which case it returns NIL.

(PROPNAMES *ATM*) [Function]

Returns a list of the property names on the property list of *ATM*.

(DEFLIST *L PROP*) [Function]

Used to put values under the same property name on the property lists of several litatoms. *L* is a list of two-element lists. The first element of each is a litatom, and the second element is the property value for the property *PROP*. Returns NIL. For example,

```
(DEFLIST '( (FOO MA) (BAR CA) (BAZ RI) ) 'STATE)
```

puts MA on FOO's STATE property, CA on BAR's STATE property, and RI on BAZ's

STATE property.

Property lists are conventionally implemented as lists of the form

(*NAME*$_1$ *VALUE*$_1$ *NAME*$_2$ *VALUE*$_2$ ···)

although the user can store anything as the property list of a litatom. However, the functions which manipulate property lists observe this convention by searching down the property lists two CDRs at a time. Most of these functions also generate an error, ARG NOT LITATOM, if given an argument which is not a litatom, so they cannot be used directly on lists. (LISTPUT, LISTPUT1, LISTGET, and LISTGET1 are functions similar to PUTPROP and GETPROP that work directly on lists. See page 2.26.) The property lists of litatoms can be directly accessed with the following functions:

(GETPROPLIST *ATM*) [Function]

Returns the property list of *ATM*.

(SETPROPLIST *ATM LST*) [Function]

If *ATM* is a non-NIL litatom, sets the property list of *ATM* to be *LST*, and returns *LST* as its value. If *ATM* is NIL, generates the error, ATTEMPT TO RPLAC NIL (unless *LST* is also NIL).

(GETLIS *X PROPS*) [Function]

Searches the property list of *X*, and returns the property list as of the first property on *PROPS* that it finds. For example,

```
← (GETPROPLIST 'X)
(PROP1 A PROP3 B A C)
← (GETLIS 'X '(PROP2 PROP3))
(PROP3 B A C)
```

Returns NIL if no element on *PROPS* is found. *X* can also be a list itself, in which case it is searched as described above. If *X* is not a litatom or a list, returns NIL.

2.4.4 Print Names

Each litatom has a print name, a string of characters that uniquely identifies that litatom. The term "print name" has been extended, however, to refer to the characters that are output when any object is printed. In Interlisp, all objects have print names, although only litatoms and strings have their print name explicitly stored. This section describes a set of functions which can be used to access and manipulate the print names of any object, though they are primarily used with the print names of litatoms.

The print name of an object is those characters that are output when the object is printed using PRIN1, e.g., the print name of the litatom ABC%(D consists of the five characters ABC(D. The print name of the list (A B C) consists of the seven characters (A B C) (two of the characters are spaces).

Sometimes we will have occasion to refer to a "PRIN2-name." The PRIN2-name of an object is those characters output when the object is printed using PRIN2. Thus the PRIN2-name of the litatom ABC%(D is the *six* characters ABC%(D. Note that the PRIN2-name depends on what readtable is being used (see page 6.32), since this determines where %'s will be inserted. Many of the functions below allow either print names or PRIN2-names to be used, as specified by *FLG* and *RDTBL* arguments. If *FLG* is NIL, print names are used. Otherwise, PRIN2-names are used, computed with respect to the readtable *RDTBL* (or the current readtable, if *RDTBL* = NIL).

Note: The print name of an integer depends on the setting of RADIX (page 6.19). The functions described in this section (UNPACK, NCHARS, etc.) define the print name of an integer as though the radix was 10, so that (PACK (UNPACK 'X9)) will always be X9 (and not sometimes X11) regardless of the setting of RADIX. However, integers will still be *printed* by PRIN1 using the current radix. The user can force these functions to use print names in the current radix by changing the setting of the variable PRXFLG (see page 6.20).

(MKATOM *x*) [Function]

Creates and returns an atom whose print name is the same as that of the string *x* or, if *x* isn't a string, the same as that of (MKSTRING *x*). Examples:

```
(MKATOM '(A B C))  =>  %(A% B% C%)

(MKATOM "1.5")  =>  1.5
```

Note that the last example returns a number, not a litatom. It is a deeply-ingrained feature of Interlisp that no litatom can have the print name of a number.

(SUBATOM *x* *N* *M*) [Function]

Equivalent to (MKATOM (SUBSTRING *x* *N* *M*)), but does not make a string pointer (see page 2.29). Returns an atom made from the *N*th through *M*th characters of the print name of *x*. If *N* or *M* are negative, they specify positions counting backwards from the end of the print name. Examples:

```
(SUBATOM "FOO1.5BAR" 4 6)  =>  1.5

(SUBATOM '(A B C) 2 -2)  =>  A% B% C
```

(PACK *x*) [Function]

If *x* is a list of atoms, PACK returns a single atom whose print name is the concatenation of the print names of the atoms in *x*. If the concatenated print name is the same as that of a number, PACK will return that number. For example,

```
(PACK '(A BC DEF G))  =>  ABCDEFG

(PACK '(1 3.4))  =>  13.4

(PACK '(1 E -2))  =>  .01
```

Although *x* is usually a list of atoms, it can be a list of arbitrary Interlisp objects. The value of PACK is still a single atom whose print name is the concatenation of the print names of all the elements of *x*, e.g.,

```
(PACK '((A B) "CD"))  =>  %(A% B%)CD
```

If *x* is not a list or NIL, PACK generates an error, ILLEGAL ARG.

(PACK* x_1 x_2 ··· x_N) [NoSpread Function]

Nospread version of PACK that takes an arbitrary number of arguments, instead of a list. Examples:,

```
(PACK* 'A 'BC 'DEF 'G)  =>  ABCDEFG

(PACK* 1 3.4)  =>  13.4
```

(UNPACK *X FLG RDTBL*) [Function]

Returns the print name of *X* as a list of single-characters atoms, e.g.,

```
(UNPACK 'ABC5D)  =>  (A B C 5 D)

(UNPACK "ABC(D")  =>  (A B C %( D)
```

If *FLG*=T, the PRIN2-name of *X* is used (computed with respect to *RDTBL*), e.g.,

```
(UNPACK "ABC(D" T)  =>  (%" A B C %( D %")

(UNPACK 'ABC%(D" T)  =>  (A B C %% %( D)
```

Note: (UNPACK *X*) performs *N* CONSes, where *N* is the number of characters in the print name of *X*.

(DUNPACK *X SCRATCHLIST FLG RDTBL*) [Function]

A destructive version of UNPACK that does not perform any CONSes but instead reuses the list *SCRATCHLIST*. If the print name is too long to fit in *SCRATCHLIST*, DUNPACK will extend it. If *SCRATCHLIST* is not a list, DUNPACK returns (UNPACK *X FLG RDTBL*).

(NCHARS *X FLG RDTBL*) [Function]

Returns the number of characters in the print name of *X*. If *FLG*=T, the PRIN2-name is used. For example,

```
(NCHARS "ABC")  =>  3

(NCHARS "ABC" T)  =>  5
```

(NTHCHAR *X N FLG RDTBL*) [Function]

Returns the *N*th character of the print name of *X* as an atom. *N* can be negative, in which case it counts from the end of the print name, e.g., -1 refers to the last character, -2 next to last, etc. If *N* is greater than the number of characters in the print name, or less than minus that number, or 0, NTHCHAR returns NIL. Examples:

```
(NTHCHAR 'ABC 2)  =>  B

(NTHCHAR 15.6 2)  =>  5

(NTHCHAR 'ABC%(D -3 T)  =>  %%

(NTHCHAR "ABC" 2)  =>  B

(NTHCHAR "ABC" 2 T)  =>  A
```

Note: NTHCHAR and NCHARS work much faster on objects that actually have an internal representation of their print name, i.e., litatoms and strings, than they do on numbers and lists, as they do not have to simulate printing.

(L-CASE *X* *FLG*) [Function]

Returns a lower case version of *X*. If *FLG* is T, the first letter is capitalized. If *X* is a string, the value of L-CASE is also a string. If *X* is a list, L-CASE returns a new list in which L-CASE is computed for each corresponding element and non-NIL tail of the original list. Examples:

```
(L-CASE 'FOO)  =>  foo

(L-CASE 'FOO T)  =>  Foo

(L-CASE "FILE NOT FOUND" T)  =>  "File not found"

(L-CASE '(JANUARY FEBRUARY (MARCH "APRIL")) T)
            =>  '(January February (March "April"))
```

(U-CASE *X*) [Function]

Similar to L-CASE, except returns the upper case version of *X*.

(U-CASEP *X*) [Function]

Returns T if *X* contains no lower case letters; NIL otherwise.

(GENSYM *CHAR*) [Function]

Returns a litatom of the form Xnnnn, where X=*CHAR* (or A if *CHAR* is NIL) and nnnn is an integer. Thus, the first one generated is A0001, the second A0002, etc. GENSYM provides a way of generating litatoms for various uses within the system.

GENNUM [Variable]

The value of GENNUM, initially 10000, determines the next GENSYM, e.g., if GENNUM is set to 10023, (GENSYM)=A0024.

The term "gensym" is used to indicate a litatom that was produced by the function GENSYM. Litatoms generated by GENSYM are the same as any other litatoms: they have property lists, and can be given function definitions. Note that the litatoms are not guaranteed to be new. For example, if the user has previously created A0012, either by typing it in, or via PACK or GENSYM itself, when GENNUM gets to 10011, the next litatom returned by GENSYM will be the A0012 already in existence.

(MAPATOMS *FN*) [Function]

Applies *FN* (a function or lambda expression) to every litatom in the system. Returns NIL.

For example,

```
(MAPATOMS (FUNCTION (LAMBDA(X)
                         (if (GETD X) then (PRINT X)]
```

will print every litatom with a function definition.

Note: In some implementations of Interlisp, unused litatoms may be garbage collected, which can effect the action of MAPATOMS.

2.4.5 Character Code Functions

Characters may be represented in two ways: as single-character atoms, or as integer character codes.[3] In many situations, it is more efficient to use character codes, so Interlisp provides parallel functions for both representations.

(PACKC *x*) [Function]

Similar to PACK except *x* is a list of character codes. For example,

```
(PACKC '(70 79 79))  =>  FOO
```

(CHCON *x FLG RDTBL*) [Function]

Like UNPACK, except returns the print name of *x* as a list of character codes. If *FLG*=T, the PRIN2-name is used. For example,

```
(CHCON 'FOO)  =>  (70 79 79)
```

(DCHCON *x SCRATCHLIST FLG RDTBL*) [Function]

Similar to DUNPACK.

(NTHCHARCODE *x N FLG RDTBL*) [Function]

Similar to NTHCHAR, except returns the character code of the *N*th character of the print name of *x*. If *N* is negative, it is interpreted as a count backwards from the end of *x*. If the absolute value of *N* is greater than the number of characters in *x*, or 0, then the value of NTHCHARCODE is NIL.

If *FLG* is T, then the PRIN2-name of *x* is used, computed with respect to the readtable *RDTBL*

(CHCON1 *x*) [Function]

Returns the character code of the first character of the print name of *x*; equal to (NTHCHARCODE *x* 1).

(CHARACTER *N*) [Function]

N is a character code. Returns the atom having the corresponding single character as its print name.

```
(CHARACTER 70)  =>  F
```

(FCHARACTER *N*) [Function]

Fast version of CHARACTER that compiles open.

The following function makes it possible to gain the efficiency that comes from dealing with character codes without losing the symbolic advantages of character atoms:

(CHARCODE *C*) [NLambda Function]

Returns the character code structure specified by *C* (unevaluated). If *C* is a 1-character atom or string, the corresponding character code is simply returned.

[3] Interlisp-D uses an 8-bit character set, so the legal character codes range from 0 to 255. Interlisp-10 uses standard 7-bit ASCII, so the range is 0-127.

Thus, (CHARCODE A) is 65, (CHARCODE 0) is 48. If *c* is a list structure, the value is a copy of *c* with all the leaves replaced by the corresponding character codes. For instance, (CHARCODE (A (B C))) => (65 (66 67))

CHARCODE permits easy specification of non-printable ASCII character codes: A multi-character litatom or string whose first character is ↑ is interpreted as the control-character corresponding to its second character. Thus, (CHARCODE ↑A) is 1, the code for control-A.

Also, if a multi-character litatom or string begins with #, this signifies a "meta-character", with a code between 128 to 255. # and ↑ may be combined, so (CHARCODE #↑A) is 129. (Note: Interlisp-10 cannot directly represent meta-characters as character litatoms, because it only supports 7-bit characters.)

The following key litatoms are mapped into the indicated codes: CR (13), LF (10), SPACE or SP (32), ESCAPE or ESC (27), BELL (7), BS (8), TAB (9), NULL (0), and DEL (127). The litatom EOL maps into the appropriate End-Of-Line character code in the different Interlisp implementations (31 in Interlisp-10, 13 in Interlisp-D, 10 in Interlisp-VAX).

Finally, CHARCODE maps NIL into NIL. This is included because some character-code producing functions sometimes return NIL (e.g. NTHCHARCODE); a test for that value can be included in a CHARCODE list along with true character-code values.

Charcode of litatomic arguments can be used wherever a structure of character codes would be appropriate. For example:

```
(FMEMB (NTHCHARCODE X 1) (CHARCODE (CR LF SPACE)))
(EQ (BIN FOO) (CHARCODE ↑C))
```

There is a macro for CHARCODE which causes the character-code structure to be constructed at compile-time. Thus, the compiled code for these examples is exactly as efficient as the less readable:

```
(FMEMB (NTHCHARCODE X 1) (QUOTE (13 10 32)))
(EQ (BIN FOO) 3)
```

(SELCHARQ *E* *CLAUSE*$_1$ ··· *CLAUSE*$_N$ *DEFAULT*) [NLambda NoSpread Function]

Similar to SELECTQ (page 4.2), except that the selection keys are determined by applying CHARCODE (instead of QUOTE) to the key-expressions. If the value of *E* is a character code or NIL and it is EQ or MEMB to the result of applying CHARCODE to the first element of a clause, the remaining forms of that clause are evaluated. Otherwise, the default is evaluated.

Thus

```
(SELCHARQ (BIN FOO)
   ((SPACE TAB) (FUM))
   ((↑D NIL) (BAR))
   (a (BAZ))
   (ZIP))
```

is exactly equivalent to

```
(SELECTQ (BIN FOO)
   ((32 9) (FUM))
   ((4 NIL) (BAR))
   (97 (BAZ))
   (ZIP))
```

Furthermore, SELCHARQ has a macro such that it always compiles as an equivalent SELECTQ.

2.5 LISTS

One of the most useful datatypes in Interlisp is the list cell, a data structure which contains pointers to two other objects, known as the CAR and the CDR of the list cell (after the accessing functions). Very complicated structures can be built out of list cells, including lattices and trees, but list cells are most frequently used for representing simple linear lists of objects.

The following functions are used to manipulate list cells:

(CONS *X* *Y*) [Function]

CONS is the primary list construction function. It creates and returns a new list cell containing pointers to *X* and *Y*. If *Y* is a list, this returns a list with *X* added at the beginning of *Y*.

(CAR *X*) [Function]

Returns the first element of the list *X*. CAR of NIL is always NIL. For all other nonlists (e.g., litatoms, numbers, strings, arrays), the value is undefined (and in some implementations may generate an error).

(CDR *X*) [Function]

Returns all but the first element of the list *X*. CDR of NIL is always NIL. The value of CDR is undefined for other nonlists.

Often, combinations of the CAR and CDR functions are used to extract various components of complex list structures. Functions of the form C···R may be used for some of these combinations:

```
(CAAR X)  ==>  (CAR (CAR X))

(CADR X)  ==>  (CAR (CDR X))

(CDDDDR X)  ==>  (CDR (CDR (CDR (CDR X))))
```

All 30 combinations of nested CARs and CDRs up to 4 deep are included in the system.

(RPLACD *X* *Y*) [Function]

Replaces the CDR of the list cell *X* with *Y*. This physically changes the internal structure of *X*, as opposed to CONS, which creates a new list cell. It is possible to construct a circular list by using RPLACD to place a pointer to the beginning of a list in a spot at the end of the list.

The value of RPLACD is *x*. An attempt to RPLACD NIL will cause an error, ATTEMPT TO RPLAC NIL (except for (RPLACD NIL NIL)). An attempt to RPLACD any other non-list will cause an error, ARG NOT LIST.

(RPLACA *x y*) [Function]

Similar to RPLACD, but replaces the CAR of *x* with *y*. The value of RPLACA is *x*. An attempt to RPLACA NIL will cause an error, ATTEMPT TO RPLAC NIL, (except for (RPLACA NIL NIL)). An attempt to RPLACA any other non-list will cause an error, ARG NOT LIST.

(RPLNODE *x a d*) [Function]

Performs (RPLACA *x a*), (RPLACD *x d*), and returns *x*.

(RPLNODE2 *x y*) [Function]

Performs (RPLACA *x* (CAR *y*)), (RPLACD *x* (CDR *y*)) and returns *x*.

(FRPLACD *x y*) [Function]
(FRPLACA *x y*) [Function]
(FRPLNODE *x a d*) [Function]
(FRPLNODE2 *x y*) [Function]

Faster versions of RPLACD, etc.

Warning: In Interlisp-10 and Interlisp-VAX, these functions compile open with no error checks on the type of *x*, so a compiled FRPLACD can produce unpredictable effects.

Usually, single list cells are not manipulated in isolation, but in structures known as "lists". By convention, a list is represented by a list cell whose CAR is the first element of the list, and whose CDR is the rest of the list (usually another list cell or the "empty list," NIL). List elements may be any Interlisp objects, including other lists.

The input syntax for a list is a sequence of Interlisp data objects (litatoms, numbers, other lists, etc.) enclosed in parentheses or brackets. Note that ( ) is read as the litatom NIL. A right bracket can be used to match all left parenthesis back to the last left bracket, or terminate the lists, e.g. (A (B (C].

If there are two or more elements in a list, the final element can be preceded by a period delimited on both sides, indicating that CDR of the final list cell in the list is to be the element immediately following the period, e.g. (A . B) or (A B C . D), otherwise CDR of the last list cell in a list will be NIL. Note that a list does not have to end in NIL. It is simply a structure composed of one or more list cells. The input sequence (A B C . NIL) is equivalent to (A B C), and (A B . (C D)) is equivalent to (A B C D). Note however that (A B . C D) will create a list containing the five litatoms A, B, %., C, and D.

Lists are printed by printing a left parenthesis, and then printing the first element of the list, then printing a space, then printing the second element, etc. until the final list cell is reached. The individual elements of a list are printed by PRIN1 if the list is being printed by PRIN1, and by PRIN2 if the list is being printed by PRINT or PRIN2. Lists are considered to terminate when CDR of some node is not a list. If CDR of this terminal node is NIL (the usual case), CAR of the terminal node is printed followed by a right parenthesis. If CDR of the terminal node is *not* NIL, CAR of the terminal node is printed, followed by a space, a period, another space, CDR of the terminal node, and then the right parenthesis. Note that a list input as (A B C . NIL) will print as (A B C), and a list input as (A B . (C D)) will print as (A B C D). Note also that PRINTLEVEL affects the printing of lists (page 6.18), and that carriage returns may be inserted where dictated by LINELENGTH (page 6.8).

Note: One must be careful when testing the equality of list structures. EQ will be true only when the two lists are the *exact* same list. For example,

```
← (SETQ A '(1 2))
(1 2)
← (SETQ B A)
(1 2)
← (EQ A B)
T
← (SETQ C '(1 2))
(1 2)
← (EQ A C)
NIL
← (EQUAL A C)
T
```

In the example above, the values of A and B are the exact same list, so they are EQ. However, the value of C is a totally different list, although it happens to have the same elements. EQUAL should be used to compare the elements of two lists. In general, one should notice whether list manipulation functions use EQ or EQUAL for comparing lists. This is a frequent source of errors.

Interlisp provides an extensive set of list manipulation functions:

2.5.1 Creating Lists

(MKLIST *x*) [Function]

"Make List." If *x* is a list or NIL, returns *x*; Otherwise, returns (LIST *x*).

(LIST x_1 x_2 ⋯ x_N) [NoSpread Function]

Returns a list of its arguments, e.g.

```
(LIST 'A 'B '(C D))  =>  (A B (C D))
```

(APPEND x_1 x_2 ⋯ x_N) [NoSpread Function]

Copies the top level of the list x_1 and appends this to a copy of the top level of the list x_2 appended to ⋯ appended to x_N, e.g.,

```
(APPEND '(A B) '(C D E) '(F G))  =>  (A B C D E F G)
```

Note that only the first N-1 lists are copied. However N=1 is treated specially; (APPEND X) copies the top level of a single list. To copy a list to all levels, use COPY.

The following examples illustrate the treatment of non-lists:

```
(APPEND '(A B C) 'D)  =>  (A B C . D)

(APPEND 'A '(B C D))  =>  (B C D)
```

```
(APPEND '(A B C . D) '(E F G))  =>  (A B C E F G)

(APPEND '(A B C . D))  =>  (A B C . D)
```

(NCONC x_1 x_2 ··· x_N) [NoSpread Function]

Returns the same value as APPEND, but actually modifies the list structure of x_1 ··· x_{n-1}.

Note that NCONC cannot change NIL to a list:

```
←(SETQ FOO NIL)
NIL
←(NCONC FOO '(A B C))
(A B C)
←FOO
NIL
```

Although the value of the NCONC is (A B C), FOO has *not* been changed. The "problem" is that while it is possible to alter list structure with RPLACA and RPLACD, there is no way to change the non-list NIL to a list.

(NCONC1 *LST* *X*) [Function]

(NCONC *LST* (LIST *X*))

(ATTACH *X* *L*) [Function]

"Attaches" *X* to the front of *L* by doing a RPLACA and RPLACD. The value is EQUAL to (CONS *X* *L*), but EQ to *L*, which it physically changes (except if *L* is NIL). (ATTACH X NIL) is the same as (CONS X NIL). Otherwise, if *L* is not a list, an error is generated, ARG NOT LIST.

2.5.2 Building Lists From Left to Right

(TCONC *PTR* *X*) [Function]

TCONC is similar to NCONC1; it is useful for building a list by adding elements one at a time at the end. Unlike NCONC1, TCONC does not have to search to the end of the list each time it is called. Instead, it keeps a pointer to the end of the list being assembled, and updates this pointer after each call. This can be considerably faster for long lists. The cost is an extra list cell, *PTR*. (CAR *PTR*) is the list being assembled, (CDR *PTR*) is (LAST (CAR *PTR*)). TCONC returns *PTR*, with its CAR and CDR appropriately modified.

PTR can be initialized in two ways. If *PTR* is NIL, TCONC will create and return a *PTR*. In this case, the program must set some variable to the value of the first call to TCONC. After that, it is unnecessary to reset the variable, since TCONC physically changes its value. Example:

```
←(SETQ FOO (TCONC NIL 1))
((1) 1)
←(for I from 2 to 5 do (TCONC FOO I))
NIL
←FOO
```

```
((1 2 3 4 5) 5)
```

If *PTR* is initially (NIL), the value of TCONC is the same as for *PTR*=NIL. but TCONC changes *PTR*. This method allows the program to initialize the TCONC variable before adding any elements to the list. Example:

```
←(SETQ FOO (CONS))
(NIL)
←(for I from 1 to 5 do (TCONC FOO I))
NIL
←FOO
((1 2 3 4 5) 5)
```

(LCONC *PTR X*) [Function]

Where TCONC is used to add *elements* at the end of a list, LCONC is used for building a list by adding *lists* at the end, i.e., it is similar to NCONC instead of NCONC1. Example:

```
←(SETQ FOO (CONS))
(NIL)
←(LCONC FOO '(1 2))
((1 2) 2)
←(LCONC FOO '(3 4 5))
((1 2 3 4 5) 5)
←(LCONC FOO NIL)
((1 2 3 4 5) 5)
```

LCONC uses the same pointer conventions as TCONC for eliminating searching to the end of the list, so that the same pointer can be given to TCONC and LCONC interchangeably. Therefore, continuing from above,

```
←(TCONC FOO NIL)
((1 2 3 4 5 NIL) NIL)
←(TCONC FOO '(3 4 5))
((1 2 3 4 5 NIL (3 4 5)) (3 4 5))
```

The functions DOCOLLECT and ENDCOLLECT also permit building up lists from left-to-right like TCONC, but without the overhead of an extra list cell. The list being maintained is kept as a circular list. DOCOLLECT adds items; ENDCOLLECT replaces the tail with its second argument, and returns the full list.

(DOCOLLECT *ITEM LST*) [Function]

"Adds" *ITEM* to the end of *LST*. Returns the new circular list. Note that *LST* is modified, but it is not EQ to the new list. The new list should be stored and used as *LST* to the next call to DOCOLLECT.

(ENDCOLLECT *LST TAIL*) [Function]

Takes *LST*, a list returned by DOCOLLECT, and returns it as a non-circular list, adding *TAIL* as the terminating CDR.

Here is an example using DOCOLLECT and ENDCOLLECT. HPRINT is used to print the results because they are circular lists. Notice that FOO has to be set to the value of DOCOLLECT as each element is added.

```
←(SETQ FOO NIL]
NIL
←(HPRINT (SETQ FOO (DOCOLLECT 1 FOO]
↑(1 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 2 FOO]
↑(2 1 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 3 FOO]
↑(3 1 2 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 4 FOO]
↑(4 1 2 3 . {1})
←(SETQ FOO (ENDCOLLECT FOO 5]
(1 2 3 4 . 5)
```

2.5.3 Copying Lists

(COPY *x*) [Function]

Creates and returns a copy of the list *x*. All levels of *x* are copied down to non-lists, so that if *x* contains arrays and strings, the copy of *x* will contain the same arrays and strings, not copies. COPY is recursive in the CAR direction only, so very long lists can be copied.

Note: To copy just the *top level* of *x*, do (APPEND *x*).

(COPYALL *x*) [Function]

Like COPY except copies down to atoms. Arrays, hash-arrays, strings, user data types, etc., are all copied. Analagous to EQUALALL (page 2.3). Note that this will not work if given a data structure with circular pointers; in this case, use HCOPYALL.

(HCOPYALL *x*) [Function]

Similar to COPYALL, except that it will work even if the data structure contains circular pointers.

2.5.4 Extracting Tails of Lists

(TAILP *x* *y*) [Function]

Returns *x*, if *x* is a *tail* of the list *y*; otherwise NIL. *x* is a tail of *y* if it is EQ to 0 or more CDRs of *y*.

Note: If *x* is EQ to 1 or more CDRs of *y*, *x* is called a "proper tail."

(NTH *x* *n*) [Function]

Returns the tail of *x* beginning with the *n*th element. Returns NIL if *x* has fewer than *n* elements. Examples:

```
(NTH '(A B C D) 1)  =>  (A B C D)
```

```
(NTH '(A B C D) 3)  =>  (C D)
```

```
(NTH '(A B C D) 9)  =>  NIL
```

```
(NTH '(A . B) 2)  =>  B
```

For consistency, if *N*=0, NTH returns (CONS NIL *x*):

```
(NTH '(A B) 0)  =>  (NIL A B)
```

(FNTH *x* *N*) [Function]

Faster version of NTH that terminates on a null-check.

(Interlisp-10) Interpreted, generates an error, BAD ARGUMENT - FNTH, if *x* ends in other than NIL.

(LAST *x*) [Function]

Returns the last list cell in the list *x*. Returns NIL if *x* is not a list. Examples:

```
(LAST '(A B C))  =>  (C)
```

```
(LAST '(A B . C))  =>  (B . C)
```

```
(LAST 'A)  =>  NIL
```

(FLAST *x*) [Function]

Faster version of LAST that terminates on a null-check.

(Interlisp-10) Interpreted, generates an error, BAD ARGUMENT - FLAST, if *x* ends in other than NIL.

(NLEFT *L* *N* *TAIL*) [Function]

NLEFT returns the tail of *L* that contains *N* more elements than *TAIL*. If *L* does not contain *N* more elements than *TAIL*, NLEFT returns NIL. If *TAIL* is NIL or not a tail of *L*, NLEFT returns the last *N* list cells in *L*. NLEFT can be used to work backwards through a list. Example:

```
←(SETQ FOO '(A B C D E))
(A B C D E)
←(NLEFT FOO 2)
(D E)
←(NLEFT FOO 1 (CDDR FOO))
(B C D E)
←(NLEFT FOO 3 (CDDR FOO))
NIL
```

(LASTN *L* *N*) [Function]

Returns (CONS X Y), where Y is the last *N* elements of *L*, and X is the initial segment, e.g.,

```
(LASTN '(A B C D E) 2)  =>  ((A B C) D E)
```

```
(LASTN '(A B) 2)  =>  (NIL A B)
```

Returns NIL if *L* is not a list containing at least *N* elements.

2.5.5 Counting List Cells

(LENGTH *x*) [Function]

Returns the length of the list *x*, where "length" is defined as the number of CDRs required to reach a non-list. Examples:

```
(LENGTH '(A B C))  =>  3

(LENGTH '(A B C . D))  =>  3

(LENGTH 'A)  =>  0
```

(FLENGTH *x*) [Function]

Faster version of LENGTH that terminates on a null-check.

(Interlisp-10) Interpreted, generates an error, BAD ARGUMENT - FLENGTH, if *x* ends in other than NIL.

(EQLENGTH *x* *N*) [Function]

Equivalent to (EQUAL (LENGTH *x*) *N*), but more efficient, because EQLENGTH stops as soon as it knows that *x* is longer than *N*. Note that EQLENGTH is safe to use on (possibly) circular lists, since it is "bounded" by *N*.

(COUNT *x*) [Function]

Returns the number of list cells in the list *x*. Thus, COUNT is like a LENGTH that goes to all levels. COUNT of a non-list is 0. Examples:

```
(COUNT '(A))  =>  1

(COUNT '(A . B))  =>  1

(COUNT '(A (B) C))  =>  4
```

In this last example, the value is 4 because the list (A *x* C) uses 3 list cells for any object *x*, and (B) uses another list cell.

(COUNTDOWN *x* *N*) [Function]

Counts the number of list cells in *x*, decrementing *N* for each one. Stops and returns *N* when it finishes counting, or when *N* reaches 0. COUNTDOWN can be used on circular structures since it is "bounded" by *N*. Examples:

```
(COUNTDOWN '(A) 100)  =>  99

(COUNTDOWN '(A . B) 100)  =>  99

(COUNTDOWN '(A (B) C) 100)  =>  96

(COUNTDOWN '(DOCOLLECT 1 NIL) 100)  =>  0
```

(EQUALN *X Y DEPTH*) [Function]

Similar to EQUAL, for use with (possibly) circular structures. Whenever the depth of CAR recursion plus the depth of CDR recursion exceeds *DEPTH*, EQUALN does not search further along that chain, and returns the litatom ?. If recursion never exceeds *DEPTH*, EQUALN returns T if the expressions *X* and *Y* are EQUAL; otherwise NIL.

```
(EQUALN '(((A)) B) '(((Z)) B) 2)  =>  ?

(EQUALN '(((A)) B) '(((Z)) B) 3)  =>  NIL

(EQUALN '(((A)) B) '(((A)) B) 3)  =>  T
```

2.5.6 Logical Operations

(LDIFF *X Y Z*) [Function]

Y must be a tail of *X*, i.e., EQ to the result of applying some number of CDRs to *X*. (LDIFF *X Y*) returns a list of all elements in *X* up to *Y*.

If *Z* is not NIL, the value of LDIFF is effectively (NCONC *Z* (LDIFF *X Y*)), i.e., the list difference is added at the end of *Z*.

If *Y* is not a tail of *X*, LDIFF generates an error, LDIFF: NOT A TAIL. LDIFF terminates on a null-check, so it will go into an infinite loop if *X* is a circular list and *Y* is not a tail.

Example:

```
←(SETQ FOO '(A B C D E F))
(A B C D E F)
←(CDDR FOO)
(C D E F)
←(LDIFF FOO (CDDR FOO))
(A B)
←(LDIFF FOO (CDDR FOO) '(1 2))
(1 2 A B)
←(LDIFF FOO '(C D E F))
LDIFF: not a tail
(C D E F)
```

Note that the value of LDIFF is always new list structure unless *Y*=NIL, in which case the value is *X* itself.

(LDIFFERENCE *X Y*) [Function]

"List Difference." Returns a list of those elements in *X* that are not members of *Y*.

(INTERSECTION *X Y*) [Function]

Returns a list whose elements are members of both lists *X* and *Y*. Note that (INTERSECTION X X) gives a list of all members of X without any duplications.

(UNION *X* *Y*) [Function]

Returns a (new) list consisting of all elements included on either of the two original lists. It is more efficient to make *X* be the shorter list.

The value of UNION is *Y* with all elements of *X* not in *Y* CONSed on the front of it. Therefore, if an element appears twice in *Y*, it will appear twice in (UNION *X* *Y*). Since (UNION '(A) '(A A)) = (A A), while (UNION '(A A) '(A)) = (A), UNION is non-commutative.

2.5.7 Searching Lists

(MEMB *X* *Y*) [Function]

Determines if *X* is a member of the list *Y*. If there is an element of *Y* EQ to *X*, returns the tail of *Y* starting with that element. Otherwise, returns NIL. Examples:

```
(MEMB 'A '(A (W) C D))  =>  (A (W) C D)

(MEMB 'C '(A (W) C D))  =>  (C D)

(MEMB 'W '(A (W) C D))  =>  NIL

(MEMB '(W) '(A (W) C D))  =>  NIL
```

(FMEMB *X* *Y*) [Function]

Faster version of MEMB that terminates on a null-check.

(Interlisp-10) Interpreted, FMEMB gives an error, BAD ARGUMENT - FMEMB, if *Y* ends in a non-list other than NIL.

(MEMBER *X* *Y*) [Function]

Identical to MEMB except that it uses EQUAL instead of EQ to check membership of *X* in *Y*. Examples:

```
(MEMBER 'C '(A (W) C D))  =>  (C D)

(MEMBER 'W '(A (W) C D))  =>  NIL

(MEMBER '(W) '(A (W) C D))  =>  ((W) C D)
```

(EQMEMB *X* *Y*) [Function]

Returns T if either *X* is EQ to *Y*, or else *Y* is a list and *X* is an FMEMB of *Y*.

2.5.8 Substitution Functions

(SUBST *NEW* *OLD* *EXPR*) [Function]

Returns the result of substituting *NEW* for all occurrences of *OLD* in the expression *EXPR*. Substitution occurs whenever *OLD* is EQUAL to CAR of some subexpression of *EXPR*, or when *OLD* is atomic and EQ to a non-NIL CDR of some subexpression of *EXPR*. For example:

```
(SUBST 'A 'B '(C B (X . B)))  =>  (C A (X . A))

(SUBST 'A '(B C) '((B C) D B C))
         =>  (A D B C)  not  (A D . A)
```

SUBST returns a copy of *EXPR* with the appropriate changes. Furthermore, if *NEW* is a list, it is copied at each substitution.

(DSUBST *NEW OLD EXPR*) [Function]

Similar to SUBST, except it does not copy *EXPR*, but changes the list structure *EXPR* itself. Like SUBST, DSUBST substitutes with a copy of *NEW*. More efficient than SUBST.

(LSUBST *NEW OLD EXPR*) [Function]

Like SUBST except *NEW* is substituted as a segment of the list *EXPR* rather than as an element. For instance,

```
(LSUBST '(A B) 'Y '(X Y Z))  =>  (X A B Z)
```

Note that if *NEW* is not a list, LSUBST returns a copy of *EXPR* with all *OLD*'s deleted:

```
(LSUBST NIL 'Y '(X Y Z))  =>  (X Z)
```

(SUBLIS *ALST EXPR FLG*) [Function]

ALST is a list of pairs:

((OLD_1 . NEW_1) (OLD_2 . NEW_2) ··· (OLD_N . NEW_N))

Each OLD_i is an atom. SUBLIS returns the result of substituting each NEW_i for the corresponding OLD_i in *EXPR*, e.g.,

```
(SUBLIS '((A . X) (C . Y)) '(A B C D))  =>  (X B Y D)
```

If *FLG*=NIL, new structure is created only if needed, so if there are no substitutions, the value is EQ to *EXPR*. If *FLG*=T, the value is always a copy of *EXPR*.

(DSUBLIS *ALST EXPR FLG*) [Function]

Similar to SUBLIS, except it does not copy *EXPR*, but changes the list structure *EXPR* itself.

(SUBPAIR *OLD NEW EXPR FLG*) [Function]

Similar to SUBLIS, except that elements of *NEW* are substituted for corresponding atoms of *OLD* in *EXPR*, e.g.,

```
(SUBPAIR '(A C) '(X Y) '(A B C D))  =>  (X B Y D)
```

As with SUBLIS, new structure is created only if needed, or if *FLG*=T, e.g., if *FLG*=NIL and there are no substitutions, the value is EQ to *EXPR*.

If *OLD* ends in an atom other than NIL, the rest of the elements on *NEW* are substituted for that atom. For example, if *OLD*=(A B . C) and *NEW*=(U V X Y Z), U is substituted for A, V for B, and (X Y Z) for C. Similarly, if *OLD* itself is an atom (other than NIL), the entire list *NEW* is substituted for it. Examples:

```
(SUBPAIR '(A B . C) '(W X Y Z) '(C A B B Y))  =>  ((Y Z) W X
X Y)
```

Note that SUBST, DSUBST, and LSUBST all substitute copies of the appropriate expression, whereas SUBLIS, and DSUBLIS, and SUBPAIR substitute the identical structure (unless *FLG*=T). For example:

```
← (SETQ FOO '(A B))
(A B)
← (SETQ BAR '(X Y Z))
(X Y Z)
← (DSUBLIS (LIST (CONS 'X FOO)) BAR)
((A B) Y Z)
← (DSUBLIS (LIST (CONS 'Y FOO)) BAR T)
((A B) (A B) Z)
← (EQ (CAR BAR) FOO)
T
← (EQ (CADR BAR) FOO)
NIL
```

2.5.9 Association Lists and Property Lists

(ASSOC *KEY ALST*) [Function]

ALST is a list of lists. ASSOC returns the first sublist of *ALST* whose CAR is EQ to *KEY*. If such a list is not found, ASSOC returns NIL. Example:

```
(ASSOC 'B '((A . 1) (B . 2) (C . 3)))  =>  (B . 2)
```

(FASSOC *KEY ALST*) [Function]

Faster version of ASSOC that terminates on a null-check.

(Interlisp-10) Interpreted, FASSOC gives an error if *ALST* ends in a non-list other than NIL, BAD ARGUMENT - FASSOC.

(SASSOC *KEY ALST*) [Function]

Same as ASSOC but uses EQUAL instead of EQ when searching for *KEY*.

(PUTASSOC *KEY VAL ALST*) [Function]

Searches *ALST* for a sublist CAR of which is EQ to *KEY*. If one is found, the CDR is replaced (using RPLACD) with *VAL*. If no such sublist is found, (CONS *KEY VAL*) is added at the end of *ALST*. Returns *VAL*. If *ALST* is not a list, generates an error, ARG NOT LIST.

Note that the argument order for ASSOC, PUTASSOC, etc. is different from that of LISTGET, LISTPUT, etc.

(LISTGET *LST PROP*) [Function]

Similar to GETPROP (page 2.7) but works on lists using property list format. Searches *LST* two elements at a time, by CDDR, looking for an element EQ to *PROP*. If one is found, returns the next element of *LST*, otherwise NIL. Returns

NIL if *LST* is not a list. Example:

```
(LISTGET '(A 1 B 2 C 3) 'B)  =>  2

(LISTGET '(A 1 B 2 C 3) 'W)  =>  NIL
```

(LISTPUT *LST PROP VAL*) [Function]

Similar to PUTPROP. Searches *LST* two elements at a time, by CDDR, looking for an element EQ to *PROP*. If *PROP* is found, replaces the next element of *LST* with *VAL*. Otherwise, *PROP* and *VAL* are added to the end of *LST*. If *LST* is a list with an odd number of elements, or ends in a non-list other than NIL, *PROP* and *VAL* are added at its beginning. Returns *VAL*. If *LST* is not a list, generates an error, ARG NOT LIST.

(LISTGET1 *LST PROP*) [Function]

Like LISTGET, but searches *LST* one CDR at a time, i.e., looks at each element. Returns the next element after *PROP*. Examples:

```
(LISTGET1 '(A 1 B 2 C 3) 'B)  =>  2

(LISTGET1 '(A 1 B 2 C 3) '1)  =>  B

(LISTGET1 '(A 1 B 2 C 3) 'W)  =>  NIL
```

Note: LISTGET1 used to be called GET.

(LISTPUT1 *LST PROP VAL*) [Function]

Like LISTPUT, except searches *LST* one CDR at a time. Returns the modified *LST*. Example:

```
←(SETQ FOO '(A 1 B 2))
(A 1 B 2)
←(LISTPUT FOO 'B 3)
(A 1 B 3)
←(LISTPUT FOO 'C 4)
(A 1 B 3 C 4)
←(LISTPUT FOO 1 'W)
(A 1 W 3 C 4)
←FOO
(A 1 W 3 C 4)
```

Note that if *LST* is not a list, no error is generated. However, since a non-list cannot be changed into a list, *LST* is not modified. In this case, the value of LISTPUT1 should be saved. Example:

```
←(SETQ FOO NIL)
NIL
←(LISTPUT FOO 'A 5)
(A 5)
←FOO
NIL
```

2.5.10 Other List Functions

(REMOVE *X* *L*) [Function]

Removes all top-level occurrences of *X* from list *L*, returning a copy of *L* with all elements EQUAL to *X* removed. Example:

```
(REMOVE 'A '(A B C (A) A))  =>  (B C (A))

(REMOVE '(A) '(A B C (A) A))  =>  (A B C A)
```

(DREMOVE *X* *L*) [Function]

Similar to REMOVE, but uses EQ instead of EQUAL, and actually modifies the list *L* when removing *X*, and thus does not use any additional storage. More efficient than REMOVE.

Note that DREMOVE cannot *change* a list to NIL:

```
←(SETQ FOO '(A))
(A)
←(DREMOVE 'A FOO)
NIL
←FOO
(A)
```

The DREMOVE above returns NIL, and does not perform any CONSes, but the value of FOO is *still* (A), because there is no way to change a list to a non-list. See NCONC.

(REVERSE *L*) [Function]

Reverses (and copies) the top level of a list, e.g.,

```
(REVERSE '(A B (C D)))  =>  ((C D) B A)
```

If *L* is not a list, REVERSE just returns *L*.

(DREVERSE *L*) [Function]

Value is the same as that of REVERSE, but DREVERSE destroys the original list *L* and thus does not use any additional storage. More efficient than REVERSE.

2.6 STRINGS

A string is an object which represents a sequence of characters. Interlisp provides functions for creating strings, concatenating strings, and creating sub-strings of a string.

The input syntax for a string is a double quote ("), followed by a sequence of any characters except double quote and %, terminated by a double quote. The % and double quote characters may be included in a string by preceding them with the escape character %.

Strings are printed by PRINT and PRIN2 with initial and final double quotes, and %s inserted where necessary for it to read back in properly. Strings are printed by PRIN1 without the delimiting double quotes and extra %s.

A "null string" containing no characters is input as "". The null string is printed by PRINT and PRIN2 as "". (PRIN1 "") doesn't print anything.

Strings are created by MKSTRING, ALLOCSTRING, SUBSTRING, and CONCAT.

Internally a string is stored in two parts; a "string pointer" and the sequence of characters. Several string pointers may reference the same character sequence, so a substring can be made by creating a new string pointer, without copying any characters. It is not possible to directly access a character sequence, so functions that refer to "strings" actually manipulate string pointers. In most cases, the user does not have to be aware of string pointers, but there are some situations where it is important to understand them. For example, suppose that *X* is a string pointer to a sequence of characters, and *Y* is another string pointer to a substring of *X*'s characters. If the characters of *Y* are modified (with RPLSTRING or RPLCHARCODE), the corresponding characters of *X* will be modified too.

(STREQUAL *X* *Y*) [Function]

Returns T if *X* and *Y* are both strings and they contain the same sequence of characters, otherwise NIL. EQUAL uses STREQUAL. Note that strings may be STREQUAL without being EQ. For instance,

```
(STREQUAL "ABC" "ABC")  =>  T

(EQ "ABC" "ABC")  =>  NIL
```

STREQUAL returns T if *X* and *Y* are the same string pointer, or two different string pointers which point to the same character sequence, or two string pointers which point to different character sequences which contain the same characters. Only in the first case would *X* and *Y* be EQ.

(ALLOCSTRING *N* *INITCHAR* *OLD*) [Function]

Creates a string of length *N* charaters of *INITCHAR* (which can be either a character code or something coercible to a character). If *INITCHAR* is NIL, it defaults to character code 0. if *OLD* is supplied, it must be a string pointer, which is re-used.

(MKSTRING *X* *FLG* *RDTBL*) [Function]

If *X* is a string, returns *X*. Otherwise, creates and returns a string containing the print name of *X*. Examples:

```
(MKSTRING "ABC")  =>  "ABC"

(MKSTRING '(A B C))  =>  "(A B C)"

(MKSTRING NIL)  =>  "NIL"
```

Note that the last example returns the string "NIL", not the atom NIL.

If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable *RDTBL*. For example,

```
(MKSTRING "ABC" T)  =>  "%"ABC%""
```

(SUBSTRING *X N M OLDPTR*) [Function]

Returns the substring of *X* consisting of the *N*th through *M*th characters of *X*. If *M* is NIL, the substring contains the *N*th character thru the end of *X*. *N* and *M* can be negative numbers, which are interpreted as counts back from the end of the string, as with NTHCHAR (page 2.10). SUBSTRING returns NIL if the substring is not well defined, e.g., *N* or *M* specify character positions outside of *X*, or *N* corresponds to a character in *X* to the right of the character indicated by *M*). Examples:

```
(SUBSTRING "ABCDEFG" 4 6)  =>  "DEF"

(SUBSTRING "ABCDEFG" 3 3)  =>  "C"

(SUBSTRING "ABCDEFG" 3 NIL)  =>  "CDEFG"

(SUBSTRING "ABCDEFG" 4 -2)  =>  "DEF"

(SUBSTRING "ABCDEFG" 6 4)  =>  NIL

(SUBSTRING "ABCDEFG" 4 9)  =>  NIL
```

If *X* is not a string, it is converted to one. For example,

```
(SUBSTRING '(A B C) 4 6)  =>  "B C"
```

SUBSTRING does not actually copy any characters, but simply creates a new string pointer to the characters in *X*. If *OLDPTR* is a string pointer, it is modified and returned.

(GNC *X*) [Function]

"Get Next Character." Returns the next character of the string *X* (as an atom); also removes the character from the string, by changing the string pointer. Returns NIL if *X* is the null string. If *X* isn't a string, a string is made. Used for sequential access to characters of a string. Example:

```
←(SETQ FOO "ABCDEFG")
"ABCDEFG"
←(GNC FOO)
A
←(GNC FOO)
B
←FOO
"CDEFG"
```

Note that if *A* is a substring of B, (GNC A) does not remove the character from B. GNC doesn't physically change the string of characters, just the string pointer.

(GLC *X*) [Function]

"Get Last Character." Returns the last character of the string *X* (as an atom); also removes the character from the string. Similar to GNC. Example:

```
←(SETQ FOO "ABCDEFG")
"ABCDEFG"
←(GLC FOO)
```

```
G
←(GLC FOO)
F
←FOO
"ABCDE"
```

(CONCAT x_1 x_2 ··· x_N) [NoSpread Function]

Returns a new string which is the concatenation of (copies of) its arguments. Any arguments which are not strings are transformed to strings. Examples:

```
(CONCAT "ABC" 'DEF "GHI")  =>  "ABCDEFGHI"

(CONCAT '(A B C) "ABC")  =>  "(A B C)ABC"
```

(CONCAT) returns the null string, "".

(CONCATLIST *x*) [Function]

x is a list of strings and/or other objects. The objects are transformed to strings if they aren't strings. Returns a new string which is the concatenation of the strings. Example:

```
(CONCATLIST '(A B (C D) "EF"))  =>  "AB(C D)EF"
```

(RPLSTRING *X N Y*) [Function]

Replaces the characters of string *X* beginning at character position *N* with string *Y*. *X* and *Y* are converted to strings if they aren't already. *N* may be positive or negative, as with SUBSTRING. Characters are smashed into (converted) *X*. Returns the string *X*. Examples:

```
(RPLSTRING "ABCDEF" -3 "END")  =>  "ABCEND"

(RPLSTRING "ABCDEFGHIJK" 4 '(A B C))  =>  "ABC(A B C)K"
```

Generates an error if there is not enough room in *X* for *Y*, i.e., the new string would be longer than the original. If *Y* was not a string, *X* will already have been modified since RPLSTRING does not know whether *Y* will "fit" without actually attempting the transfer.

Note that if *X* is a substring of Z, Z will also be modified by the action of RPLSTRING. Example:

```
← (SETQ FOO "ABCDEFG")
"ABCDEFG"
← (SETQ BAR (SUBSTRING FOO 4 6)
"DEF"
← (RPLSTRING BAR 2 "XY")
"DXY"
← FOO
"ABCDXYG"
```

(RPLCHARCODE *X N CHARCODE*) [Function]

Replaces the *N*th character of the string *X* with the character code *CHARCODE*. *N* may be positive or negative. Returns the new *X*. Similar to RPLSTRING. Example:

```
(RPLCHARCODE "ABCDE" 3 (CHARCODE F))  =>  "ABFDE"
```

(STRPOS *PAT STRING START SKIP ANCHOR TAIL*) [Function]

STRPOS is a function for searching one string looking for another. *PAT* and *STRING* are both strings (or else they are converted automatically). STRPOS searches *STRING* beginning at character number *START*, (or 1 if *START* is NIL) and looks for a sequence of characters equal to *PAT*. If a match is found, the character position of the first matching character in *STRING* is returned, otherwise NIL. Examples:

```
(STRPOS "ABC" "XYZABCDEF")  =>  4

(STRPOS "ABC" "XYZABCDEF" 5)  =>  NIL

(STRPOS "ABC" "XYZABCDEFABC" 5)  =>  10
```

SKIP can be used to specify a character in *PAT* that matches any character in *STRING*. Examples:

```
(STRPOS "A&C&" "XYZABCDEF" NIL '&)  =>  4

(STRPOS "DEF&" "XYZABCDEF" NIL '&)  =>  NIL
```

If *ANCHOR* is T, STRPOS compares *PAT* with the characters beginning at position *START* (or 1 if *START* is NIL). If that comparison fails, STRPOS returns NIL without searching any further down *STRING*. Thus it can be used to compare one string with some *portion* of another string. Examples:

```
(STRPOS "ABC" "XYZABCDEF" NIL NIL T)  =>  NIL

(STRPOS "ABC" "XYZABCDEF" 4 NIL T)  =>  4
```

Finally, if *TAIL* is T, the value returned by STRPOS if successful is not the starting position of the sequence of characters corresponding to *PAT*, but the position of the first character after that, i.e., the starting position plus (NCHARS *PAT*). Examples:

```
(STRPOS "ABC" "XYZABCDEFABC" NIL NIL NIL T)  =>  7

(STRPOS "A" "A" NIL NIL NIL T)  =>  2
```

If *TAIL*=NIL, STRPOS returns NIL, or a character position within *STRING* which can be passed to SUBSTRING. In particular, (STRPOS "" "") => NIL. However, if *TAIL*=T, STRPOS may return a character position outside of *STRING*. For instance, note that the second example above returns 2, even though "A" has only one character.

(STRPOSL *A STR START NEG*) [Function]

STR is a string (or else it is converted automatically to a string), *A* is a list of characters or character codes. STRPOSL searches *STR* beginning at character number *START* (or else 1 if *START*=NIL) for one of the characters in *A*. If one is found, STRPOSL returns as its value the corresponding character position, otherwise NIL. Example:

```
(STRPOSL '(A B C) "XYZBCD")  =>  4
```

If *NEG*=T, STRPOSL searches for a character *not* on *A*. Example:

```
(STRPOSL '(A B C) "ABCDEF" NIL T)  =>  4
```

If any element of *A* is a number, it is assumed to be a character code. Otherwise, it is converted to a character code via CHCON1. Therefore, it is more efficient to call STRPOSL with *A* a list of character *codes*.

If *A* is a bit table, it is used to specify the characters (see MAKEBITTABLE below)

STRPOSL uses a "bit table" data structure to search efficiently. If *A* is not a bit table, it is converted it to a bit table using MAKEBITTABLE. If STRPOSL is to be called frequently with the same list of characters, a considerable savings can be achieved by converting the list to a bit table *once*, and then passing the bit table to STRPOSL as its first argument.

(MAKEBITTABLE *L* *NEG* *A*) [Function]

Returns a bit table suitable for use by STRPOSL. *L* is a list of characters or character codes, *NEG* is the same as described for STRPOSL. If *A* is a bit table, MAKEBITTABLE modifies and returns it. Otherwise, it will create a new bit table.

Note: if *NEG*=T, STRPOSL must call MAKEBITTABLE whether *A* is a list *or* a bit table. To obtain bit table efficiency with *NEG*=T, MAKEBITTABLE should be called with *NEG*=T, and the resulting "inverted" bit table should be given to STRPOSL with *NEG*=NIL.

2.7 ARRAYS

An array in Interlisp is an object representing a one-dimensional vector of objects. Arrays do not have input syntax; they can only be created by the function ARRAY. Arrays are printed by PRINT, PRIN2, and PRIN1 as # followed by an integer.

Note: Interlisp-10 and Interlisp-Vax provide a much more primitive version of arrays than other implementations of Interlisp. See page 2.33.

(ARRAY *SIZE* *TYPE* *INIT* *ORIG*) [Function]

Creates and returns a new array capable of containing *SIZE* objects of type *TYPE*. *TYPE* may be one of BIT, BYTE, WORD, FIXP, FLOATP, POINTER, or DOUBLEPOINTER.[4] ARRAY also accepts any "type" which is legal in DATATYPE records (such as (BITS 7), FLAG, see page 3.7). (Note: DATATYPE types are coerced into the next "enclosing" array type. Therefore, users should not rely on truncation of values stored in arrays of these types.)

[4] For backward compatibility with Interlisp-10 arrays, *TYPE* can be NIL or 0 (meaning to create an array of type DOUBLEPOINTER) or *SIZE* (meaning an array of type FIXP). For arrays of type DOUBLEPOINTER, the functions ELTD and SETD are defined the same as in Interlisp-10 (page 2.34). For arrays of any other type, ELTD and SETD are the same as ELT and SETD. Combined POINTER/FIXP arrays are not supported. Interlisp-D users should avoid using Interlisp-10 arrays.

INIT is the initial value in each element of the new array. If not specified, the array elements will be initialized with 0 (for number arrays) or NIL (all other types).

Arrays can have either 0-origin or 1-origin indexing, as specified by the *ORIG* argument; if *ORIG* is not specified, the default is 1.

(ELT *A* *N*) [Function]

Returns the *N*th element of the array *A*.

(SETA *A* *N* *V*) [Function]

Sets the *N*th element of the array *A* to *V*. SETA returns *V*.

(ARRAYTYP *A*) [Function]

Returns a value corresponding to the second argument to ARRAY.

Note: If ARRAY coerced the array type as described above, ARRAYTYP will return the *new* type.

(ARRAYSIZE *A*) [Function]

Returns the size of array *A*. Generates the error, ARG NOT ARRAY, if *A* is not an array.

(ARRAYORIG *A*) [Function]

Returns the origin of array *A*, which may be 0 or 1. Generates an error, ARG NOT ARRAY, if *A* is not an array.

(COPYARRAY *A*) [Function]

Returns a new array of the same size and type as *A*, and with the same contents as *A*. Generates an ARG NOT ARRAY error, if *A* is not an array.

2.7.1 Interlisp-10 Arrays

Interlisp-10 and Interlisp-Vax have a more primitive array facility than the other implementations of Interlisp. In Interlisp-10, arrays are partitioned into four sections: a header, a section containing unboxed numbers, a section containing list cells (each with a CAR and CDR), and a section containing relocation information. The last three sections can each be of arbitrary length (including 0); the header is two words long and contains the length of the other sections. The unboxed number region of an array is used to store 36 bit quantities that are not Interlisp pointers, and therefore are not to be chased during garbage collections, e.g. machine instructions. The relocation informaion is used when the array contains the definition of a compiled function, and specifies which locations in the *unboxed* region of the array must be changed if the array is moved during a garbage collection.

ARRAY returns an "array pointer" to the beginning of the array, but it is also possible to create a pointer into the middle of an array. ARRAYP will accept a pointer into the middle of an array, but ELT, SETA, ELTD, and SETD generate an error, ARG NOT ARRAY, if *A* is not an array pointer to the beginning of an array.

Array-pointers print as #*NNNN*, where *NNNN* is the octal representation of the pointer. Note that #*NNNN* will be read as a literal atom, and not an array pointer.

The following functions are used to manipulate Interlisp-10 arrays:

(ARRAY *N P V*) [Function]

Allocates a block of *N*+2 words, of which the first two are header information. The next *P* ($\leq$ *N*) words contain unboxed numbers, and are initialized to unboxed 0. The last *N*-*P* ($\geq$ 0) words are list cells; both CAR and CDR are available for storing information, and each is initialized to *V*. If *P* is NIL, 0 is used (i.e., an array containing all Interlisp pointers). ARRAY returns an "array pointer" to the array.

If sufficient space is not available for the array, a garbage collection of array space is initiated. If this is unsuccessful in obtaining sufficient space, an error is generated, ARRAYS FULL.

(ELT *A N*) [Function]

Returns the *N*th element of the array *A*. (ELT *A* 1) is the first element of the array (actually corresponds to the 3rd cell because of the 2 word header).

If *N* corresponds to the unboxed number region of *A*, ELT returns the full 36 bit word as a boxed integer. If *N* corresponds to the list cell region of *A*, ELT returns the CAR of the corresponding element.

(SETA *A N V*) [Function]

Sets the *N*th element of the array *A* to *V*. If *N* corresponds to the unboxed number region of *A*, *V* must be a number, and is unboxed and stored as a full 36 bit word into the *N*th element of *A*. If *N* corresponds to the list cell region of *A*, *V* replaces the CAR of the *N*th element. SETA returns *V*.

(ELTD *A N*) [Function]

Same as ELT for the unboxed number region of *A*, but returns the CDR of the *N*th element, if *N* corresponds to the list cell region of *A*.

(SETD *A N V*) [Function]

Same as SETA for the unboxed number region of *A*, but sets the CDR half of the *N*th element, if *N* corresponds to the list cell region of *A*. SETD returns *V*.

(ARRAYTYP *A*) [Function]

Returns the number of unboxed number words of array *A*. This value corresponds to the second argument to ARRAY.

(ARRAYP *X*) [Function]

Returns *X* if *X* is an array pointer, otherwise NIL. No check is made to ensure that *X* actually addresses the *beginning* of an array.

(ARRAYBEG *A*) [Function]

If *A* is a pointer into the middle of an array, returns the pointer to its beginning. Otherwise returns NIL.

(ARRAYORIG *A*) [Function]

Returns 1. A dummy function provided for compatibility with other Interlisp arrays.

2.8 HASH ARRAYS

Hash arrays provide a mechanism for associating arbitrary lisp objects ("hash keys") with other objects ("hash values"), such that the hash value associated with a particular hash key can be quickly obtained. A set of associations could be represented as a list or array of pairs, but these schemes are very inefficient when the number of associations is large. There are functions for creating hash arrays, putting a hash key/value pair in a hash array, and quickly retrieving the hash value associated with a given hash key.

Hash keys can be any lisp object, but is should be noted that the hash array functions use EQ for comparing hash keys. Therefore, if non-atoms are used as hash keys, the exact same object (not a copy) must be used to retrieve the hash value.

In the description of the functions below, the argument *HARRAY* has one of three forms: NIL, in which case a hash array provided by the system, SYSHASHARRAY, is used; a hash-array created by the function HARRAY; or a list, CAR of which is a hash array. The latter form is used for specifying what is to be done on overflow, as described below.

(HARRAY *LEN*) [Function]

Creates a hash array containing at least *LEN* hash keys.

(HARRAYSIZE *HARRAY*) [Function]

Returns the size of *HARRAY*; the number of hash keys it can hold before becoming "full".

(CLRHASH *HARRAY*) [Function]

Clears all hash keys/values from *HARRAY*. Returns *HARRAY*.

(PUTHASH *KEY VAL HARRAY*) [Function]

Associates the hash value *VAL* with the hash key *KEY* in *HARRAY*. Replaces the previous hash value, if any. If *VAL* is NIL, any old association is removed (hence a hash value of NIL is not allowed). If *HARRAY* is full when PUTHASH is called with a key not already in the hash array, the function HASHOVERFLOW is called, and the PUTHASH is done to the value returned (see below). Returns *VAL*.

(GETHASH *KEY HARRAY*) [Function]

Returns the hash value associated with the hash key *KEY* in *HARRAY*. Returns NIL, if *KEY* is not found.

(REHASH *OLDHARRAY NEWHARRAY*) [Function]

Hashes all hash keys and values in *OLDHARRAY* into *NEWHARRAY*. The two hash arrays do not have to be (and usually aren't) the same size. Returns *NEWHARRAY*.

(MAPHASH *HARRAY MAPHFN*) [Function]

MAPHFN is a function of two arguments. For each hash key in *HARRAY*, *MAPHFN* will be applied to (1) the hash value, and (2) the hash key. For example,

```
[MAPHASH A
   (FUNCTION (LAMBDA (VAL KEY)
                     (if (LISTP KEY) then (PRINT VAL)]
```

will print the hash value for all hash keys that are lists. MAPHASH returns *HARRAY*.

(DMPHASH *HARRAY*$_1$ *HARRAY*$_2$ ··· *HARRAY*$_N$) [NLambda NoSpread Function]

Prints on the primary output file LOADable forms which will restore the hash-arrays contained as the values of the atoms *HARRAY*$_1$, *HARRAY*$_2$, ··· *HARRAY*$_N$. Example: (DMPHASH SYSHASHARRAY) will dump the system hash-array.

Note: all EQ identities except atoms and small integers are lost by dumping and loading because READ will create new structure for each item. Thus if two lists contain an EQ substructure, when they are dumped and loaded back in, the corresponding substructures while EQUAL are no longer EQ. The HORRIBLEVARS file package command (page 11.25) provides a way of dumping hash tables such that these identities are preserved.

2.8.1 Hash Overflow

When a hash array becomes full, attempting to add another hash key will cause the function HASHOVERFLOW to be called. This will either automatically enlarge the hash array, or cause the error HASH TABLE FULL. How hash overflow is handled is determined by the form that was passed to PUTHASH:

HARRAY — If a plain hash array is passed to a hash function, and it overflows, the error HASH ARRAY FULL is generated.

NIL — If a hash function is passed NIL as its *HARRAY* argument, the system hash array SYSHASHARRAY is used. This array is not used by the system, but is provided for the user. If SYSHASHARRAY overflows, it is automatically enlarged by 1.5.

(*HARRAY* . *N*) — *N* is a positive integer. This form specifies that upon hash overflow, a new hash-array is created with *N* more cells than the current hash-array.

(*HARRAY* . *F*) — *F* is a floating point number. This form specifies that upon hash overflow, the new hash array will be *F* times the size of the current hash-array.

(*HARRAY* . *FN*) — *FN* is a function name or a lambda expression. This form specifies that upon hash overflow, *FN* is called with (*HARRAY* . *FN*) as its argument. If *FN* returns a number, the number will be the size of the new hash array. Otherwise, the new size defaults to 1.5 times the size of the old hash array. *FN* could be used to print a message, or perform some monitor function.

(*HARRAY*) — Equivalent to (*HARRAY* . 1.5).

If a list form is used, upon hash overflow the new hash-array is RPLACAed into the dotted pair, and HASHOVERFLOW returns it.

2.9 NUMBERS AND ARITHMETIC FUNCTIONS

Numerical atoms, or simply numbers, do not have value cells, function definition cells, property lists, or explicit print names. There are three different types of numbers in Interlisp: small integers, large integers, and floating point numbers. Small integers are those integers that can be directly stored within a pointer value. The range of small integers is implementation-dependent. Since a large integer or floating point number can be (in value) any full word quantity (and vice versa), it is necessary to distinguish between those full word quantities that represent large integers or floating point numbers, and other Interlisp pointers. We do this by "boxing" the number: When a large integer or floating point number is created (via an arithmetic operation or by READ), Interlisp gets a new word from "number storage" and puts the large integer or floating point number into that word. Interlisp then passes around the pointer to that word, i.e., the "boxed number", rather than the actual quantity itself. Then when a numeric function needs the actual numeric quantity, it performs the extra level of addressing to obtain the "value" of the number. This latter process is called "unboxing". Note that unboxing does not use any storage, but that each boxing operation uses one new word of number storage. Thus, if a computation creates many large integers or floating point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, or of floating point number space. Different implementations of Interlisp may use different boxing strategies. Thus, while lots of arithmetic operations *may* lead to garbage collections, this is not necessarily always the case.

The following functions can be used to distinguish the different types of numbers:

(SMALLP *x*) [Function]

Returns *x*, if *x* is a small integer; NIL otherwise. Does *not* generate an error if *x* is not a number.

(FIXP *x*) [Function]

Returns *x*, if *x* is an integer (between MIN.FIXP and MAX.FIXP); NIL otherwise. Note that FIXP is true for both large and small integers. Does *not* generate an error if *x* is not a number.

(FLOATP *x*) [Function]

Returns *x* if *x* is a floating point number; NIL otherwise. Does *not* give an error if *x* is not a number.

(NUMBERP *x*) [Function]

Returns *x*, if *x* is a number of any type (FIXP or FLOATP); NIL otherwise. Does *not* generate an error if *x* is not a number.

Note that if (NUMBERP *x*) is true, then either (FIXP *x*) or (FLOATP *x*) is true.

Each small integer has a unique representation, so EQ may be used to check equality. Note that EQ should not be used for large integers or floating point numbers, EQP, IEQP, or EQUAL must be used instead.

(EQP *x* *y*) [Function]

Returns T, if *x* and *y* are EQ, or equal numbers; NIL otherwise. Note that EQ may be used if *x* and *y* are known to be *small* integers. EQP does not convert *x* and *y* to integers, e.g., (EQP 2000 2000.3) => NIL, but it can be used to compare an integer and a floating point number, e.g., (EQP 2000 2000.0) => T. EQP does *not* generate an error if *x* or *y* are not numbers.

Note: EQP can also be used to compare stack pointers (page 7.3) and compiled code objects (page 5.8).

2.9.1 Integer Arithmetic

The input syntax for an integer is an optional sign (+ or -) followed by a sequence of digits, followed by an optional Q, and terminated by a delimiting character. If the Q is present, the digits are interpreted in octal, otherwise in decimal, e.g. 77Q and 63 both correspond to the same integers, and in fact are indistinguishable internally since no record is kept of how integers were created.

The setting of RADIX (page 6.19), determines how integers are printed: signed or unsigned, octal or decimal.

Integers are created by PACK and MKATOM when given a sequence of characters observing the above syntax, e.g. (PACK '(1 2 Q)) => 10. Integers are also created as a result of arithmetic operations.

The range of integers of various types is implementation-dependent. This information is accessable to the user through the following variables:

MIN.SMALLP [Variable]
MAX.SMALLP [Variable]

The smallest/largest possible small integer.

MIN.FIXP [Variable]
MAX.FIXP [Variable]

The smallest/largest possible large integer.

MIN.INTEGER [Variable]
MAX.INTEGER [Variable]

The smallest/largest possible integer representable. Currently, these variables are equal to MIN.FIXP and MAX.FIXP; they may be different in future implementations with other methods for representing integers.

In Interlisp-D, the action taken on integer overflow is determined with the following function:

(OVERFLOW *FLG*) [Function]

Sets a flag that determines the system response to integer overflow; returns the previous setting. If *FLG*=T, an error occurs on integer overflow. If *FLG*=NIL, the largest (or smallest) integer is returned as the result of the overflowed computation. If *FLG*=0, the result is returned modulo 2↑32 (the default action).

All of the functions described below work on integers. Unless specified otherwise, if given a floating point number, they first convert the number to an integer by truncating the fractional bits, e.g., (IPLUS 2.3 3.8)=5; if given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

(IPLUS X_1 X_2 ⋯ X_N) [NoSpread Function]

Returns the sum $X_1 + X_2 + \cdots + X_N$. (IPLUS)=0.

(IMINUS *X*) [Function]

-*X*

(IDIFFERENCE *X* *Y*) [Function]

X - *Y*

(ADD1 x) [Function]

$x + 1$

(SUB1 x) [Function]

$x - 1$

(ITIMES x_1 x_2 $\cdots$ x_N) [NoSpread Function]

Returns the product $x_1 * x_2 * \cdots * x_N$. (ITIMES) = 1.

(IQUOTIENT x y) [Function]

x / y truncated. Examples:

```
(IQUOTIENT 3 2)  =>  1

(IQUOTIENT -3 2)  =>  -1
```

(IREMAINDER x y) [Function]

Returns the remainder when x is divided by y. Example:

```
(IREMAINDER 3 2)  =>  1
```

(IMOD x y) [Function]

Computes the integer modulus; this differs from IREMAINDER in that the result is always a non-negative integer in the range [0, y).

(IGREATERP x y) [Function]

T, if $x > y$; NIL otherwise.

(ILESSP x y) [Function]

T, if $x < y$; NIL otherwise.

(IGEQ x y) [Function]

T, if $x \geq y$; NIL otherwise.

(ILEQ x y) [Function]

T, if $x \leq y$; NIL otherwise.

(IMIN x_1 x_2 $\cdots$ x_N) [NoSpread Function]

Returns the minimum of x_1, x_2, $\cdots$, x_N. (IMIN) returns the largest possible large integer, the value of MAX.FIXP.

(IMAX x_1 x_2 ... x_N) [NoSpread Function]

Returns the maximum of x_1, x_2, $\cdots$, x_N. (IMAX) returns the smallest possible large integer, the value of MIN.FIXP.

(IEQP N M) [Function]

Returns T if N and M are EQ or equal integers; NIL otherwise. Note that EQ may be used if N and M are known to be *small* integers. IEQP converts N and M to integers, e.g., (IEQP 2000 2000.3) => T. Causes NON-NUMERIC ARG error if either N or M are not numbers.

(ZEROP x) [Function]

(EQ x 0).

Note: ZEROP should not be used for floating point numbers because it uses EQ. Use (EQP *x* 0) instead.

(MINUSP *x*) [Function]

Returns T if *x* is negative; NIL otherwise. Does not convert *x* to an integer, but simply checks the sign bit.

(FIX *x*) [Function]

If *x* is an integer, returns *x*. Otherwise, converts *x* to an integer by truncating fractional bits, e.g., (FIX 2.3) => 2, (FIX -1.7) => -1.

Since FIX is also a programmer's assistant command (page 8.10), typing FIX directly to Interlisp will not cause the function FIX to be called.

(GCD *x* *y*) [Function]

Returns the greatest common divisor of *x* and *y*, e.g., (GCD 72 64)=8.

2.9.2 Logical Arithmetic Functions

(LOGAND x_1 x_2 ··· x_N) [NoSpread Function]

Returns the logical AND of all its arguments, as an integer. Example:

```
(LOGAND 7 5 6)  =>  4
```

(LOGOR x_1 x_2 ··· x_N) [NoSpread Function]

Returns the logical OR of all its arguments, as an integer. Example:

```
(LOGOR 1 3 9)  =>  11
```

(LOGXOR x_1 x_2 ··· x_N) [NoSpread Function]

Returns the logical exclusive OR of its arguments, as an integer. Example:

```
(LOGXOR 11 5)  =>  14

(LOGXOR 11 5 9) <=> (LOGXOR 14 9)  =>  7
```

(LSH *x* *n*) [Function]

(arithmetic) "Left Shift." Returns *x* shifted left *n* places, with the sign bit unaffected. *x* can be positive or negative. If *n* is negative, *x* is shifted *right* -*n* places.

(RSH *x* *n*) [Function]

(arithmetic) "Right Shift." Returns *x* shifted right *n* places, with the sign bit unaffected, and copies of the sign bit shifted into the leftmost bit. *x* can be positive or negative. If *n* is negative, *x* is shifted *left* -*n* places.

Warning: Be careful if using RSH to simulate division; RSHing a negative number is not generally equivalent to deviding by a power of two.

(LLSH *x* *n*) [Function]

"Logical Left Shift."

(LRSH *X N*) [Function]

"Logical Right Shift."

(INTEGERLENGTH *N*) [Function]

Returns the number of bits needed to represent *N* (coerçed to a FIXP). This is equivalent to: 1+floor[log2[abs[*N*]]]. (INTEGERLENGTH 0) = 0.

(POWEROFTWOP *N*) [Function]

Returns non-NIL if *N* (coerced to a FIXP) is a power of two.

(EVENP *X Y*) [NoSpread Function]

If *Y* is not given, equivalent to (ZEROP (IMOD *X* 2)); otherwise equivalent to (ZEROP (IMOD *X Y*)).

(ODDP *X Y*) [NoSpread Function]

Equivalent to (NOT (EVENP *X Y*)).

The difference between a logical and arithmetic right shift lies in the treatment of the sign bit. Logical shifting treats it just like any other bit; arithmetic shifting will not change it, and will "propagate" rightward when actually shifting rightwards. Note that shifting (arithmetic) a negative number "all the way" to the right yields -1, not 0.

The following "logical" arithmetic functions are derived from Common Lisp, and have both macro and function definitions (the macros are for speed in running of compiled code). The following code equivalences are primarily for definitional purposes, and should not be considered an implementation (especially since the real implementation tends to be faster and less "consy" than would be apparent from the code here).

Note: The following logical functions are currently only implemented in Interlisp-D.

(LOGNOT *N*) [Function]

(LOGXOR *N* -1)

(BITTEST *N MASK*) [Function]

(NOT (ZEROP (LOGAND *N MASK*)))

(BITCLEAR *N MASK*) [Function]

(LOGAND *N* (LOGNOT *MASK*))

(BITSET *N MASK*) [Function]

(LOGOR *N MASK*)

(MASK.1'S *POSITION SIZE*) [Function]

```
(LLSH (SUB1 (EXPT 2 SIZE))
      POSITION)
```

(MASK.0'S *POSITION SIZE*) [Function]

(LOGNOT (MASK.1'S *POSITION SIZE*))

(LOADBYTE *N POSITION SIZE*) [Function]

(LOGAND (LRSH *N POSITION*)

```
                  (MASK.1'S 0 SIZE))
```

(DEPOSITBYTE *N POSITION SIZE BYTE*) [Function]

```
    (LOGOR (BITCLEAR N (MASK.1'S POSITION SIZE))
           (LLSH (LOGAND BYTE (MASK.1'S 0 SIZE))
                 POSITION))
```

(ROT *X N FIELDSIZE*) [Function]

"Rotate bits in field". This is a slight extension of the CommonLisp ROT function. It performs a bitwise left-rotation of the integer *X*, by *N* places, within a field of *FIELDSIZE* bits wide. Bits being shifted out of the position selected by (EXPT 2 (SUB1 *FIELDSIZE*)) will flow into the "units" position.

The optional argument *FIELDSIZE* defaults to the "cell" size (the integerlength of the current maximum FIXP), and must either be a positive integer, or else be one of the litatoms CELL or WORD. In the latter two cases the appropriate numerical values are respectively substituted. A macro optimizes the case where *FIELDSIZE* is WORD and *N* is 1.

The notions of position and size can be combined to make up a "byte specifier", which is constructed by the macro BYTE [note reversal of arguments as compare with above functions]:

(BYTE *SIZE POSITION*) [Macro]

Constructs and returns a "byte specifier" containing *SIZE* and *POSITION*.

(BYTESIZE *BYTESPEC*) [Macro]

Returns the *SIZE* componant of the "byte specifier" *BYTESPEC*.

(BYTEPOSITION *BYTESPEC*) [Macro]

Returns the *POSITION* componant of the "byte specifier" *BYTESPEC*.

(LDB *BYTESPEC VAL*) [Macro]

```
    (LOADBYTE VAL
              (BYTEPOSITION BYTESPEC)
              (BYTESIZE BYTESPEC))
```

(DPB *N BYTESPEC VAL*) [Macro]

```
    (DEPOSITBYTE VAL
                 (BYTEPOSITION BYTESPEC)
                 (BYTESIZE BYTESPEC)
                 N)
```

2.9.3 Floating Point Arithmetic

A floating point number is input as a signed integer, followed by a decimal point, followed by another sequence of digits called the fraction, followed by an exponent (represented by E followed by a signed integer) and terminated by a delimiter.

Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a floating point number from an integer. For example, the following will be recognized as floating point numbers:

```
5.      5.00    5.01    .3
5E2     5.1E2   5E-3    -5.2E+6
```

Floating point numbers are printed using the format control specified by the function FLTFMT (page 6.20). FLTFMT is initialized to T, or free format. For example, the above floating point numbers would be printed free format as:

```
5.0     5.0     5.01    .3
500.0   510.0   .005    -5.2E6
```

Floating point numbers are created by the read program when a "." or an E appears in a number, e.g., 1000 is an integer, 1000. a floating point number, as are 1E3 and 1.E3. Note that 1000D, 1000F, and 1E3D are perfectly legal literal atoms. Floating point numbers are also created by PACK and MKATOM, and as a result of arithmetic operations.

PRINTNUM (page 6.21) permits greater controls on the printed appearance of floating point numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

The floating point number range is stored in the following variables:

MIN.FLOAT [Variable]

The smallest possible floating point number.

MAX.FLOAT [Variable]

The largest possible floating point number.

All of the functions described below work on floating point numbers. Unless specified otherwise, if given an integer, they first convert the number to a floating point number, e.g., (FPLUS 1 2.3) <=> (FPLUS 1.0 2.3) => 3.3; if given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

(FPLUS X_1 X_2 $\cdots$ X_N) [NoSpread Function]

$X_1 + X_2 + \cdots + X_N$

(FMINUS X) [Function]

$- X$

(FDIFFERENCE X Y) [Function]

$X - Y$

(FTIMES X_1 X_2 $\cdots$ X_N) [NoSpread Function]

$X_1 * X_2 * \cdots * X_N$

(FQUOTIENT X Y) [Function]

X / Y

(FREMAINDER X Y) [Function]

Returns the remainder when X is divided by Y. Equivalent to:

```
(FDIFFERENCE X (FTIMES Y (FIX (FQUOTIENT X Y))))
```

Example:

```
(FREMAINDER 7.5 2.3)  =>  0.6
```

(MINUSP *X*) [Function]

T, if *X* is negative; NIL otherwise. Works for both integers and floating point numbers.

(FGREATERP *X Y*) [Function]

T, if *X* > *Y*, NIL otherwise.

(FLESSP *X Y*) [Function]

T, if *X* < *Y*, NIL otherwise.

(FEQP *X Y*) [Function]

Returns T if *N* and *M* are equal floating point numbers; NIL otherwise. FEQP converts *N* and *M* to floating point numbers.Causes NON-NUMERIC ARG error if either *N* or *M* are not numbers.

(FMIN X_1 X_2 ⋯ X_N) [NoSpread Function]

Returns the minimum of X_1, X_2, ⋯, X_N. (FMIN) returns the largest possible floating point number, the value of MAX.FLOAT.

(FMAX X_1 X_2 ... X_N) [NoSpread Function]

Returns the maximum of X_1, X_2, ⋯, X_N. (FMAX) returns the smallest possible floating point number, the value of MIN.FLOAT.

(FLOAT *X*) [Function]

Converts *X* to a floating point number. Example:

```
(FLOAT 0)  =>  0.0
```

2.9.4 Mixed Arithmetic

The functions in this section are "generic" floating point arithmetic functions. If any of the arguments are floating point numbers, they act exactly like floating point functions, and float all arguments, and return a floating point number as their value. Otherwise, they act like the integer functions. If given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

(PLUS X_1 X_2 ⋯ X_N) [NoSpread Function]

$X_1 + X_2 + \cdots + X_N$.

(MINUS *X*) [Function]

$- X$

(DIFFERENCE *X Y*) [Function]

$X - Y$

(TIMES x_1 x_2 ⋯ x_N) [NoSpread Function]

$x_1 * x_2 * \cdots * x_N$

(QUOTIENT *x* *y*) [Function]

If *x* and *y* are both integers, returns (IQUOTIENT *x* *y*), otherwise (FQUOTIENT *x* *y*).

(REMAINDER *x* *y*) [Function]

If *x* and *y* are both integers, returns (IREMAINDER *x* *y*), otherwise (FREMAINDER *x* *y*).

(GREATERP *x* *y*) [Function]

T, if $x > y$, NIL otherwise.

(LESSP *x* *y*) [Function]

T if $x < y$, NIL otherwise.

(GEQ *x* *y*) [Function]

T, if $x \geq y$, NIL otherwise.

(LEQ *x* *y*) [Function]

T, if $x \leq y$, NIL otherwise.

(MIN x_1 x_2 ⋯ x_N) [NoSpread Function]

Returns the minimum of x_1, x_2, ⋯, x_N. (MIN) returns the value of MAX.INTEGER.

(MAX x_1 x_2 ⋯ x_N) [NoSpread Function]

Returns the maximum of x_1, x_2, ⋯, x_N. (MAX) returns the value of MIN.INTEGER.

(ABS *x*) [Function]

x if $x > 0$, otherwise -*x*. ABS uses GREATERP and MINUS, (not IGREATERP and IMINUS).

2.9.5 Special Functions

(EXPT *m* *n*) [Function]

Returns *m*↑*n*. If *m* is an integer and *n* is a positive integer, returns an integer, e.g, (EXPT 3 4) => 81, otherwise returns a floating point number. If *m* is negative and *n* fractional, an error is generated, ILLEGAL EXPONENTIATION. If *n* is floating and either too large or too small, an error is generated, VALUE OUT OF RANGE EXPT.

(SQRT *n*) [Function]

Returns the square root of *n* as a floating point number. *n* may be fixed or floating point. Generates an error if *n* is negative.

(LOG *x*) [Function]

Returns the natural logarithm of *x* as a floating point number. *x* can be integer or floating point.

(ANTILOG *X*) [Function]

Returns the floating point number whose logarithm is *X*. *X* can be integer or floating point. Example:

```
(ANTILOG 1)  =  e  =>  2.71828...
```

(SIN *X RADIANSFLG*) [Function]

Returns the sine of *X* as a floating point number. *X* is in degrees unless *RADIANSFLG*=T.

(COS *X RADIANSFLG*) [Function]

Similar to SIN.

(TAN *X RADIANSFLG*) [Function]

Similar to SIN.

(ARCSIN *X RADIANSFLG*) [Function]

X is a number between -1 and 1 (or an error is generated). The value of ARCSIN is a floating point number, and is in degrees unless *RADIANSFLG*=T. In other words, if (ARCSIN *X RADIANSFLG*)=*Z* then (SIN *Z RADIANSFLG*)=*X*. The range of the value of ARCSIN is -90 to +90 for degrees, $-\pi/2$ to $\pi/2$ for radians.

(ARCCOS *X RADIANSFLG*) [Function]

Similar to ARCSIN. Range is 0 to 180, 0 to π.

(ARCTAN *X RADIANSFLG*) [Function]

Similar to ARCSIN. Range is 0 to 180, 0 to π.

(ARCTAN2 *Y X RADIANSFLG*) [Function]

Computes (ARCTAN (FQUOTIENT *Y X*) *RADIANSFLG*), and returns a corresponding value in the range -180 to 180 (or $-\pi$ to π), i.e. the result is in the proper quadrant as determined by the signs of *X* and *Y*.

(RAND *LOWER UPPER*) [Function]

Returns a pseudo-random number between *LOWER* and *UPPER* inclusive, i.e., RAND can be used to generate a sequence of random numbers. If both limits are integers, the value of RAND is an integer, otherwise it is a floating point number. The algorithm is completely deterministic, i.e., given the same initial state, RAND produces the same sequence of values. The internal state of RAND is initialized using the function RANDSET described below.

(RANDSET *X*) [Function]

Returns the internal state of RAND. If *X*=NIL, just returns the current state. If *X*=T, RAND is initialized using the clocks, and RANDSET returns the new state. Otherwise, *X* is interpreted as a previous internal state, i.e., a value of RANDSET, and is used to reset RAND. For example,

```
← (SETQ OLDSTATE (RANDSET))
...
← (for X from 1 to 10 do (PRIN1 (RAND 1 10)))
2847592748NIL
← (RANDSET OLDSTATE)
```

```
...
← (for X from 1 to 10 do (PRIN1 (RAND 1 10)))
2847592748NIL
```

CHAPTER 3

THE RECORD PACKAGE

The advantages of "data abstraction" have long been known: more readable code, fewer bugs, the ability to change the data structure without having to make major modifications to the program, etc. The record package encourages and facilitates this good programming practice by providing a uniform syntax for creating, accessing and storing data into many different types of data structures (arrays, list structures, association lists, etc.) as well as removing from the user the task of writing the various manipulation routines. The user declares (once) the data structures used by his programs, and thereafter indicates the manipulations of the data in a data-structure-independent manner. Using the declarations, the record package automatically computes the corresponding Interlisp expressions necessary to accomplish the indicated access/storage operations. If the data structure is changed by modifying the declarations, the programs automatically adjust to the new conventions.

The user describes the format of a data structure (record) by making a "record declaration" (see page 3.5). The record declaration is a description of the record, associating names with its various parts, or "fields". For example, the record declaration `(RECORD MSG (FROM TO . TEXT))` describes a data structure called `MSG`, which contains three fields: `FROM`, `TO`, and `TEXT`. The user can reference these fields by name, to retrieve their values or to store new values into them, by using the `FETCH` and `REPLACE` operators (page 3.1). The `CREATE` operator (page 3.3) is used for creating new instances of a record, and `TYPE?` (page 3.4) is used for testing whether an object is an instance of a particular record. (note: all record operators can be in either upper or lower case.)

Records may be implemented in a variety of different ways, as determined by the first element ("record type") of the record declaration. `RECORD` (used to specify elements and tails of a list structure) is just one of several record types currently implemented. The user can specify a property list format by using the record type `PROPRECORD`, or that fields are to be associated with parts of a data structure via a specified hash array by using the record type `HASHLINK`, or that an entirely new data type be allocated (as described on page 3.14) by using the record-type `DATATYPE`.

The record package is implemented through the DWIM/CLISP facilities, so it contains features such as spelling correction on field names, record types, etc. Record operations are translated using all CLISP declarations in effect (standard/fast/undoable); it is also possible to declare local record declarations that override global ones (see page 16.9).

The file package includes a `RECORDS` file package command for dumping record declarations (page 11.25), and `FILES?` and `CLEANUP` will inform the user about records that need to be dumped.

3.1 FETCH AND REPLACE

The fields of a record are accessed and changed with the `FETCH` and `REPLACE` operators. If the record `MSG` has the record declaration `(RECORD MSG (FROM TO . TEXT))`, and `X` is a `MSG` data structure, `(fetch FROM of X)` will return the value of the `FROM` field of `X`, and `(replace FROM of X with`

Y) will replace this field with the value of Y. In general, the value of a REPLACE operation is the same as the value stored into the field.

Note that the form (fetch FROM of X) implicitly states that X is an instance of the record MSG, or at least it should to be treated as such for this particular operation. In other words, the interpretation of (fetch FROM of X) never depends on the value of X. Therefore, if X is not a MSG record, this may produce incorrect results. The TYPE? record operation (page 3.4) may be used to test the types of objects.

If there is another record declaration, (RECORD REPLY (TEXT . RESPONSE)), then (fetch TEXT of X) is ambiguous, because X could be either a MSG or a REPLY record. In this case, an error will occur, AMBIGUOUS RECORD FIELD. To clarify this, FETCH and REPLACE can take a list for their "field" argument: (fetch (MSG TEXT) of X) will fetch the TEXT field of an MSG record.

Note that if a field has an *identical* interpretation in two declarations, e.g. if the field TEXT occurred in the same location within the declarations of MSG and REPLY, then (fetch TEXT of X) would *not* be considered ambiguous.

Another complication can occur if the fields of a record are themselves records. The fields of a record can be further broken down into sub-fields by a "subdeclaration" within the record declaration (see page 3.10). For example,

```
(RECORD NODE (POSITION . LABEL) (RECORD POSITION (XLOC . YLOC)))
```

permits the user to access the POSITION field with (fetch POSITION of X), or its subfield XLOC with (fetch XLOC of X).

The user may also elaborate a field by declaring that field name in a *separate* record declaration (as opposed to an embedded subdeclaration). For instance, the TEXT field in the MSG and REPLY records above may be subdivided with the seperate record declaration (RECORD TEXT (HEADER . TXT)). Fields of subfields (to any level of nested subfields) are accessed by specifying the "data path" as a list of record/field names, where there is some path from each record to the next in the list. For instance, (fetch (MSG TEXT HEADER) of X) indicates that X is to be treated as a MSG record, its TEXT field should be accessed, and *its* HEADER field should be accessed. Only as much of the data path as is necessary to disambiguate it needs to be specified. In this case, (fetch (MSG HEADER) of X) is sufficient. The record package interprets a data path by performing a tree search among all current record declarations for a path from each name to the next, considering first local declarations (if any) and then global ones. The central point of separate declarations is that the (sub)record is *not* tied to another record (as with embedded declarations), and therefore can be used in many different contexts. If a data-path rather than a single field is ambiguous, (e.g., if there were yet another declaration (RECORD TO (NAME . HEADER)) and the user specified (fetch (MSG HEADER) of X)), the error AMBIGUOUS DATA PATH is generated.

FETCH and REPLACE forms are translated using the CLISP declarations in effect. FFETCH and FREPLACE are versions which insure fast CLISP declarations will be in effect, /REPLACE insures undoable declarations.

3.2 CREATE

Record operations can be applied to arbitrary structures, i.e., the user can explicitely creating a data structure (using CONS, etc), and then manipulate it with FETCH and REPLACE. However, to be consistant with the idea of data abstraction, new data should be created using the same declarations that define its data paths. This can be done with an expression of the form:

(CREATE *RECORD-NAME* . *ASSIGNMENTS*)

A CREATE expression translates into an appropriate Interlisp form using CONS, LIST, PUTHASH, ARRAY, etc., that creates the new datum with the various fields initialized to the appropriate values. *ASSIGNMENTS* is optional and may contain expressions of the following form:

FIELD-NAME ← *FORM*
Specifies initial value for *FIELD-NAME*.

USING *FORM* Specifies that for all fields not explicitly given a value, the value of the corresponding field in *FORM* is to be used.

COPYING *FORM* Similar to USING except the corresponding values are copied (with COPYALL).

REUSING *FORM* Similar to USING, except that wherever possible, the corresponding *structure* in *FORM* is used.

SMASHING *FORM* A new instance of the record is not created at all; rather, the value of *FORM* is used and smashed.

The record package goes to great pains to insure that the order of evaluation in the translation is the same as that given in the original CREATE expression if the side effects of one expression might affect the evaluation of another. For example, given the declaration (RECORD CONS (CAR . CDR)), the expression (CREATE CONS CDR←X CAR←Y) will translate to (CONS Y X), but (CREATE CONS CDR←(FOO) CAR←(FIE)) will translate to ((LAMBDA ($$1) (CONS (PROGN (SETQ $$1 (FOO)) (FIE)) $$1))) because FOO might set some variables used by FIE.

Note that (CREATE *RECORD* REUSING *FORM* ...) does not itself do any destructive operations on the value of *FORM*. The distinction between USING and REUSING is that (CREATE *RECORD* REUSING *FORM* ...) will incorporate as much as possible of the old data structure into the new one being created, while (CREATE *RECORD* USING *FORM* ...) will create a completely new data structure, with only the contents of the fields re-used. For example, CREATE REUSING a PROPRECORD just CONSes the new property names and values onto the list, while CREATE USING copies the top level of the list. Another example of this distinction occurs when a field is elaborated by a subdeclaration: USING will create a new instance of the sub-record, while REUSING will use the old contents of the field (unless some field of the subdeclaration is assigned in the CREATE expression.)

If the value of a field is neither explicitly specified, nor implicitly specified via USING, COPYING or REUSING, the default value in the declaration is used, if any, otherwise NIL. (Note: For BETWEEN fields in DATATYPE records, N_1 is used; for other non-pointer fields zero is used.) For example, following (RECORD A (B C D) D ← 3),

```
(CREATE A B←T)  ==>  (LIST T NIL 3)

(CREATE A B←T USING X)  ==>  (LIST T (CADR X) (CADDR X))
```

```
(CREATE A B←T COPYING X))  ==>  [LIST T (COPYALL (CADR X)) (COPYALL (CADDR X]
(CREATE A B←T REUSING X)  ==>  (CONS T (CDR X))
```

3.3 TYPE?

The record package allows the user to test if a given datum "looks like" an instance of a record. This can be done via an expression of the form

(TYPE? *RECORD-NAME FORM*)

TYPE? is mainly intended for records with a record type of DATATYPE or TYPERECORD. For DATATYPEs, the TYPE? check is exact; i.e. the TYPE? expression will return non-NIL only if the value of *FORM* is an instance of the record named by *RECORD-NAME*. For TYPERECORDs, the TYPE? expression will check that the value of *FORM* is a list beginning with *RECORD-NAME*. For ARRAYRECORDs, it checks that the value is an array of the correct size. For PROPRECORDs and ASSOCRECORDs, a TYPE? expression will make sure that the value of *FORM* is a property/association list with property names among the field-names of the declaration.

Attempting to execute a TYPE? expression for a record of type ACCESSFNS, HASHLINK or RECORD will cause an error, TYPE? NOT IMPLEMENTED FOR THIS RECORD. The user can (re)define the interpretation of TYPE? expressions for a particular declaration by inclusion of an expression of the form (TYPE? *COM*) in the record declaration (see page 3.9).

3.4 WITH

Often it is necessary to manipulate the values of the fields of a particular record. The WITH construct can be used to talk about the fields of a record as if they were variables within a lexical scope:

(WITH *RECORD-NAME RECORD-INSTANCE* $FORM_1 \cdots FORM_N$)

RECORD-NAME is the name of a record, and *RECORD-INSTANCE* is an expression which evaluates to an instance of that record. The expressions $FORM_1 \cdots FORM_N$ are evaluated so that references to variables which are field-names of *RECORD-NAME* are implemented via fetch and SETQs of those variables are implemented via replace.

For example, given

```
(RECORD RECN (FLD1 FLD2))
(SETQ INST (CREATE RECN FLD1 ← 10 FLD2 ← 20))
```

Then the construct

```
(with RECN INST (SETQ FLD2 (PLUS FLD1 FLD2]
```

is equivalent to

```
(replace FLD2 of INST with (PLUS (fetch FLD1 of INST) (fetch FLD2 of INST]
```

Note that the substitution is lexical: this operates by actually doing a substitution inside the forms.

3.5 RECORD DECLARATIONS

A record is defined by evaluating a record declaration,[1] which is an expression of the form:

(*RECORD-TYPE RECORD-NAME FIELDS* . *RECORD-TAIL*)

RECORD-TYPE specifies the "type" of data being described by the record declaration, and thereby implicitly specifies how the corresponding access/storage operations are performed. *RECORD-TYPE* currently is either RECORD, TYPERECORD, ARRAYRECORD, ATOMRECORD, ASSOCRECORD, PROPRECORD, DATATYPE, HASHLINK, ARRAYBLOCK or ACCESSFNS. RECORD and TYPERECORD are used to describe list structures, DATATYPE to describe user data-types, ARRAYRECORD to describe arrays, ATOMRECORD to describe (the property list of) litatoms, PROPRECORD to describe lists in property list format, and ASSOCRECORD to describe association list format. HASHLINK can be used with any type of data: it simply specifies the data path to be a hash-link. ACCESSFNS is also type-less; the user specifies the data-paths in the record declaration itself, as described below.

RECORD-NAME is a litatom used to identify the record declaration for creating instances of the record via CREATE, testing via TYPE?, and dumping to files via the RECORDS file package command (page 11.25). DATATYPE and TYPERECORD declarations also use *RECORD-NAME* to identify the data structure (as described below).

FIELDS describes the structure of the record. Its exact interpretation varies with *RECORD-TYPE*:

RECORD [Record Type]

FIELDS is a list structure whose non-NIL literal atoms are taken as field-names to be associated with the corresponding elements and tails of a list structure. For example, with the record declaration (RECORD MSG (FROM TO . TEXT)), (fetch FROM of X) translates as (CAR X).

NIL can be used as a place marker to fill an unnamed field, e.g., (A NIL B) describes a three element list, with B corresponding to the third element. A number may be used to indicate a sequence of NILs, e.g. (A 4 B) is interpreted as (A NIL NIL NIL NIL B).

TYPERECORD [Record Type]

Similar to RECORD, except that *RECORD-NAME* is also used as an indicator in CAR of the datum to signify what "type" of record it is. This type-field is used by the record package in the translation of TYPE? expressions. CREATE will insert an extra field containing *RECORD-NAME* at the beginning of the structure, and the translation of the access and storage functions will take this extra field into

[1] Local record declarations are defined by including an expression of this form in the CLISP declaration for that function, rather than evaluating the expression itself (see page 16.10).

account. For example, for (TYPERECORD MSG (FROM TO . TEXT)), (fetch FROM of X) translates as (CADR X), not (CAR X).

ASSOCRECORD [Record Type]

FIELDS is a list of literal atoms. The fields are stored in association-list format:

((*FIELDNAME*$_1$. *VALUE*$_1$) (*FIELDNAME*$_2$. *VALUE*$_2$) ···)

Accessing is performed with ASSOC (or FASSOC, depending on current CLISP declarations), storing with PUTASSOC.

PROPRECORD [Record Type]

FIELDS is a list of literal atoms. The fields are stored in "property list" format:

(*FIELDNAME*$_1$ *VALUE*$_1$ *FIELDNAME*$_2$ *VALUE*$_2$ ···)

Accessing is performed with LISTGET, storing with LISTPUT.

Both ASSOCRECORD and PROPRECORD are useful for defining data structures in which it is often the case that many of the fields are NIL. A CREATE for these record types only stores those fields which are non-NIL. Note, however, that with the record declaration (PROPRECORD FIE (H I J)) the expression (CREATE FIE) would still construct (H NIL), since a later operation of (replace J of X with Y) could not possibly change the instance of the record if it were NIL.

ARRAYRECORD [Record Type]

FIELDS is a list of field-names that are associated with the corresponding elements of an array. NIL can be used as a place marker for an unnamed field (element). Positive integers can be used as abbreviation for the corresponding number of NILs. For example, (ARRAYRECORD (ORG DEST NIL ID 3 TEXT)) describes an eight element array, with ORG corresponding to the first element, ID to the fourth, and TEXT to the eighth.

Note that ARRAYRECORD only creates arrays of pointers. Other kinds of arrays must be implemented by the user with ACCESSFNS.

HASHLINK [Record Type]

FIELDS is either an atom *FIELD-NAME*, or a list (*FIELD-NAME* *HARRAYNAME* *HARRAYSIZE*). *HARRAYNAME* indicates the hash-array to be used; if not given, SYSHASHARRAY is used. *HARRAYSIZE* is used for initializing the hash array: if *HARRAYNAME* has not been initialized at the time of the declaration, it will be set to (LIST (HARRAY (OR *HARRAYSIZE* 100))). HASHLINKs are useful as subdeclarations to other records to add additional fields to already existing datastructures. For example, suppose that FOO is a record declared with (RECORD FOO (A B C)). To add an aditional field BAR, without modifying the already-existing data strutures, redeclare FOO with:

```
(RECORD FOO (A B C) (HASHLINK FOO (BAR BARHARRAY)))
```

Now, (fetch BAR of X) will translate into (GETHASH X BARHARRAY), hashing off the existing *list* X.

ATOMRECORD [Record Type]

FIELDS is a list of property names, e.g., (ATOMRECORD (EXPR CODE MACRO

BLKLIBRARYDEF)). Accessing is performed with GETPROP, storing with PUTPROP. As with ACCESSFNS, CREATE is not initially defined for ATOMRECORD records.

DATATYPE [Record Type]

Specifies that a new user data type with type name *RECORD-NAME* be allocated via DECLAREDATATYPE (page 3.14). Unlike other record-types, the records of a DATATYPE declaration are represented with a completely new Interlisp type, and not in terms of other existing types.

FIELDS is a list of field specifications, where each specification is either a list (*FIELDNAME* . *FIELDTYPE*), or an atom *FIELDNAME*. If *FIELDTYPE* is omitted, it defaults to POINTER. Options for *FIELDTYPE* are:

POINTER Field contains a pointer to any arbitrary Interlisp object.

BITS *N* Field contains an *N*-bit unsigned integer.

BETWEEN N_1 N_2 A generalization of BITS. Field may contain an integer *X*, such that *X* is greater than or equal to N_1 and less than or equal to N_2. Enough bits are allocated to store a number between 0 and N_2-N_1; N_1 is appropriately added or subtracted when the field is accessed or stored into.

INTEGER or FIXP Field contains a full word signed integer (the size is implementation-dependent).

FLOATING or FLOATP

Field contains a full word floating point number.

FLAG Field is a one bit field that "contains" T or NIL.

For example, the declaration

```
(DATATYPE FOO
   ((FLG BITS 12)
    TEXT
    (CNT BETWEEN 10 25)
    HEAD
    (DATE BITS 18)
    (PRIO FLOATP)
    (READ? FLAG)))
```

would define a data type FOO which occupies (in Interlisp-10) three words of storage with two pointer fields (one word), a full word floating point number, fields for an 18, 12, and 4 bit unsigned integer, and a flag (one bit), with 1 bit left over. Fields are allocated in such a way as to optimize the storage used and not necessarily in the order specified. To store this information in a conventional RECORD list structure, e.g., (RECORD MSG (FLG TEXT CNT DATE PRIO . HEAD)), would take 5 words of list space and up to three number boxes (for FLG, DATE, and PRIO).

Since the user data type must be set up at *run*-time, the RECORDS file package command will dump a DECLAREDATATYPE expression as well as the DATATYPE declaration itself. The INITRECORDS file package command (page 11.25) will dump only the DECLAREDATATYPE expression.

Note: DATATYPE declarations should be used with caution within local declarations, since a new and different data type is allocated for each one with a different name.

ARRAYBLOCK [Record Type]

(Not implemented in Interlisp-D) Similar to a DATATYPE declaration, except that the objects it creates and manipulates are arrays. As with DATATYPE's, the actual order of the fields of the ARRAYBLOCK may be shuffled around in order to satisfy garbage collector constraints.

For example,

```
(ARRAYBLOCK FOO
   ((F1 INTEGER)
    (F2 FLOATING)
    (F3 POINTER)
    (F4 BETWEEN -30 -2)
    (F5 BITS 12)
    (F6 FLAG)))
```

ACCESSFNS [Record Type]

FIELDS is a list of elements of the form (*FIELD-NAME ACCESSDEF SETDEF*), i.e. for each fieldname, the user specifies how it is to be accessed and set. *ACCESSDEF* should be a function of one argument, the datum, and will be used for accessing. *SETDEF* should be a function of two arguments, the datum and the new value, and will be used for storing. *SETDEF* may be omitted, in which case, no storing operations are allowed. *ACCESSDEF* and/or *SETDEF* may also be a LAMBDA expression or a form written in terms of variables DATUM and (in *SETDEF*) NEWVALUE. For example, given the declaration

```
[ACCESSFNS ((FIRSTCHAR (NTHCHAR DATUM 1)
                       (RPLSTRING DATUM 1 NEWVALUE))
            (RESTCHARS (SUBSTRING DATUM 2]
```

(replace FIRSTCHAR of X with Y) would translate to (RPLSTRING X 1 Y). Since no *SETDEF* is given for the RESTCHARS field, attempting to perform (replace RESTCHARS of X with Y) would generate an error, REPLACE UNDEFINED FOR FIELD. Note that ACCESSFNS do not have a CREATE definition. However, the user may supply one in the defaults and/or subdeclarations of the declaration, as described below. Attempting to CREATE an ACCESSFNS record without specifying a create definition will cause an error CREATE NOT DEFINED FOR THIS RECORD.

ACCESSDEF and *SETDEF* can also be a property list which specify FAST, STANDARD and UNDOABLE versions of the ACCESSFNS forms, e.g.

```
[ACCESSFNS LITATOM ((DEF (STANDARD GETD FAST FGETD)
                         (STANDARD PUTD UNDOABLE /PUTD]
```

means if FAST declaration is in effect, use FGETD for fetching, if UNDOABLE, use

/PUTD for saving.

The ACCESSFNS facility allows the use of data-structures not specified by one of the built-in record types. For example, one possible representation of a data-structure is to store the fields in *parallel* arrays, especially if the number of instances required is known, and they do not need to be garbage collected. Thus, to implement a data structure called LINK with two fields FROM and TO, one would have two arrays FROMARRAY and TOARRAY. The representation of an "instance" of the record would be an integer which is used to index into the arrays. This can be accomplished with the declaration:

```
[ACCESSFNS LINK
   ((FROM (ELT FROMARRAY DATUM)
          (SETA FROMARRAY DATUM NEWVALUE))
    (TO (ELT TOARRAY DATUM)
        (SETA TOARRAY DATUM NEWVALUE)))
   (CREATE (PROG1 (SETQ LINKCNT (ADD1 LINKCNT))
                  (SETA FROMARRAY LINKCNT FROM)
                  (SETA TOARRAY LINKCNT TO)))
   (INIT (PROGN (SETQ FROMARRAY (ARRAY 100))
                (SETQ FROMARRAY (ARRAY 100))]
```

To CREATE a new LINK, a counter is incremented and the new elements stored (although the CREATE form given the declaration should actually include a test for overflow).

RECORD-TAIL is optional. It may contain expressions of the form:

FIELD-NAME ← *FORM*

Allows the user to specify within the record declaration the default value to be stored in *FIELD-NAME* by a CREATE (if no value is given within the CREATE expression itself). Note that *FORM* is evaluated at CREATE time, not when the declaration is made.

(CREATE *FORM*)

Defines the manner in which CREATE of this record should be performed. This provides a way of specifying how ACCESSFNS should be created or overriding the usual definition of CREATE. If *FORM* contains the field-names of the declaration as variables, the forms given in the CREATE operation will be substituted in. If the word DATUM appears in the create form, the *original* CREATE definition is inserted. This effectively allows the user to "advise" the create.

Note: (CREATE *FORM*) may also be specified as "*RECORD-NAME* ← *FORM*", e.g. C ← (CONS A D).

(INIT *FORM*)

Specifies that *FORM* should be evaluated when the record is declared. *FORM* will also be dumped by the INITRECORDS file package command (page 11.25).

For example, see the example of an ACCESSFNS record declaration above. In this example, FROMARRAY and TOARRAY are initialized with an INIT form.

(TYPE? *FORM*)

Defines the manner in which TYPE? expressions are to be translated. *FORM* may either be an expression in terms of DATUM or a function of one argument.

Note: (TYPE? *FORM*) may also be specified as "*RECORD-NAME* @ *FORM*", e.g. C @ LISTP.

(SUBRECORD *NAME* . *DEFAULTS*)

NAME must be a field that appears in the current declaration and the name of another record. This says that, for the purposes of translating CREATE expressions, substitute the top-level declaration of *NAME* for the SUBRECORD form, adding on any defaults specified.

For example: Given (RECORD B (E F G)), (RECORD A (B C D) (SUBRECORD B)) would be treated like (RECORD A (B C D) (RECORD B (E F G))) for the purposes of translating CREATE expressions.

a subdeclaration (i.e., a record declaration.)

The *RECORD-NAME* of a subdeclaration must be either the *RECORD-NAME* of its immediately superior declaration or one of the superior's field-names. Instead of identifying the declaration as with top level declarations, the record-name of a subdeclaration identifies the parent field or record that is being described by the subdeclaration. Subdeclarations can be nested to an arbitrary depth.

Giving a subdeclaration (RECORD $NAME_1$ $NAME_2$) is a simple way of defining a *synonym* for the field $NAME_1$.

Note that, in a few cases, it makes sense for a given field to have more than one subdeclaration. For example, in

```
(RECORD (A . B) (PROPRECORD B (FOO FIE FUM)) (HASHLINK B C))
```

B is elaborated by both a PROPRECORD and a HASHLINK. Similarly,

```
(RECORD (A B) (RECORD A (C D)) (RECORD A (FOO FIE)))
```

is also acceptable, and essentially "overlays" (FOO FIE) and (C D), i.e. (fetch FOO of X) and (fetch C of X) would be equivalent. In such cases, the *first* subdeclaration is the one used by CREATE.

3.6 DEFINING NEW RECORD TYPES

In addition to the built-in record types, users can declare their own record types by performing the following steps:

(1) Add the new record-type to the value of CLISPRECORDTYPES;.

(2) Perform (MOVD 'RECORD *RECORD-TYPE*), i.e. give the record-type the same definition as that of the function RECORD;

(3) Put the name of a function which will return the translation on the property list of *RECORD-TYPE*, as the value of the property USERRECORDTYPE. Whenever a record declaration of type *RECORD-TYPE* is encountered, this function will be passed the record declaration as its argument, and should return a *new* record declaration which the record package will then use in its place.

3.7 RECORD MANIPULATION FUNCTIONS

The user may edit (or delete) global record declarations with the function:

(EDITREC *NAME* $COM_1 \cdots COM_N$) [NLambda NoSpread Function]

Nospread nlambda function similar to EDITF or EDITV. EDITREC calls the editor on a copy of all declarations in which *NAME* is the record-name or a field name. On exit, it redeclares those that have changed and undeclares any that have been deleted. If *NAME* is NIL, *all* declarations are edited.

$COM_1 \cdots COM_N$ are (optional) edit commands.

When the user redeclares a global record, the translations of all expressions involving that record or any of its fields are automatically deleted from CLISPARRAY, and thus will be recomputed using the new information. If the user changes a *local* record declaration, or changes some other CLISP declaration, e.g., STANDARD to FAST, and wishes the new information to affect record expressions already translated, he must make sure the corresponding translations are removed, usually either by CLISPIFYing or applying the !DW edit macro.

(RECLOOK *RECORDNAME* — — — —) [Function]

Returns the entire declaration for the record named *RECORDNAME*; NIL if no record declaration with name *RECORDNAME*. Note that the record package maintains internal state about current record declarations, so performing destructive operations (e.g. NCONC) on the value of RECLOOK may leave the record package in an inconsistant state. To change a record declaration, use EDITREC.

(FIELDLOOK *FIELDNAME*) [Function]

Returns the list of declarations in which *FIELDNAME* is the name of a field.

(RECORDFIELDNAMES *RECORDNAME*) [Function]

Returns the list of fields declared in record *RECORDNAME*. *RECORDNAME* may either be a name or an entire declaration.

(RECORDACCESS *FIELD DATUM DEC TYPE NEWVALUE*) [Function]

TYPE is one of FETCH, REPLACE, FFETCH, FREPLACE, /REPLACE or their lowercase equivalents. *TYPE*=NIL means FETCH. If *TYPE* corresponds to a fetch operation, i.e. is FETCH, or FFETCH, RECORDACCESS performs (*TYPE FIELD* OF *DATUM*). If *TYPE* corresponds to a replace, RECORDACCESS performs (*TYPE FIELD* OF *DATUM* WITH *NEWVALUE*). *DEC* is an optional declaration; if given, *FIELD* is interpreted as a field name of that declaration.

Note that RECORDACCESS is relatively inefficient, although it is better than constructing the equivalent form and performing an EVAL.

3.8 CHANGETRAN

A very common programming construction consists of assigning a new value to some datum that is a function of the current value of that datum. Some examples of such read-modify-write sequences include:

(SETQ X (IPLUS X 1))	Incrementing a counter
(SETQ X (CONS Y X))	Pushing an item on the front of a list
(PROG1 (CAR X) (SETQ X (CDR X)))	Popping an item off a list

It is easier to express such computations when the datum in question is a simple variable as above than when it is an element of some larger data structure. For example, if the datum to be modified was (CAR X), the above examples would be:

```
(CAR (RPLACA X (IPLUS (CAR X) 1)))

(CAR (RPLACA X (CONS Y (CAR X)))

(PROG1 (CAAR X) (RPLACA X (CDAR X)))
```

and if the datum was an element in an array, (ELT A N), the examples would be:

```
(SETA A N (IPLUS (ELT A N) 1)))

(SETA A N (CONS Y (ELT A N))))

(PROG1 (CAR (ELT A N)) (SETA A N (CDR (ELT A N))))
```

The difficulty in expressing (and reading) modification idioms is in part due to the well-known assymmetry of setting versus accessing operations on structures: RPLACA is used to smash what CAR would return, SETA corresponds to ELT, and so on.

The "Changetran" facility is designed to provide a more satisfactory notation in which to express certain common (but user-extensible) structure modification operations. Changetran defines a set of CLISP words that encode the kind of modification that is to take place, e.g. pushing on a list, adding to a number, etc. More important, the expression that indicates the datum whose value is to be modified needs to be stated only once. Thus, the "change word" ADD is used to increase the value of a datum by the sum of a set of numbers. Its arguments are an expression denoting the datum, and a set of items to be added to its current value. The datum expression must be a variable or an accessing expression (envolving fetch, CAR, LAST, ELT, etc) that can be translated to the appropriate setting expression.

For example, (ADD (CADDR X) (FOO)) is equivalent to:

```
(CAR (RPLACA (CDDR X)
             (PLUS (FOO) (CADDR X)))
```

If the datum expression is a complicated form involving subsidiary function calls, such as (ELT (FOO X) (FIE Y))), Changetran goes to some lengths to make sure that those subsidiary functions are evaluated only once (it binds local variables to save the results), even though they logically appear in both the setting and accessing parts of the translation. Thus, in thinking about both efficiency and possible side effects, the user can rely on the fact that the forms will be evaluated only as often as they appear in the expression.

For ADD and all other changewords, the lower-case version (add, etc.) may also be specified. Like other CLISP words, change words are translated using all CLISP declarations in effect (see page 16.9).

The following is a list of those change words recognized by Changetran. Except for POP, the value of all built-in changeword forms is defined to be the new value of the datum.

(ADD *DATUM* $ITEM_1$ $ITEM_2$ ···) [Change Word]

Adds the specified items to the current value of the datum, stores the result back in the datum location. The translation will use IPLUS, PLUS, or FPLUS according to the CLISP declarations in effect.

(PUSH *DATUM* $ITEM_1$ $ITEM_2$ ···) [Change Word]

CONSes the items onto the front of the current value of the datum, and stores the result back in the datum location. For example, (PUSH X A B) would translate as (SETQ X (CONS A (CONS B X))).

(PUSHNEW *DATUM* *ITEM*) [Change Word]

Like PUSH (with only one item) except that the item is not added if it is already FMEMB of the datum's value.

Note that, whereas (CAR (PUSH X 'FOO)) will always be FOO, (CAR (PUSHNEW X 'FOO)) might be something else if FOO already existed in the middle of the list.

(PUSHLIST *DATUM* $ITEM_1$ $ITEM_2$ ···) [Change Word]

Similar to PUSH, except that the items are APPENDed in front of the current value of the datum. For example, (PUSHLIST X A B) would translate as (SETQ X (APPEND A B X)).

(POP *DATUM*) [Change Word]

Returns CAR of the current value of the datum after storing its CDR into the datum. The current value is computed only once even though it is referenced twice. Note that this is the only built-in changeword for which the value of the form is not the new value of the datum.

(SWAP $DATUM_1$ $DATUM_2$) [Change Word]

Sets $DATUM_1$ to $DATUM_2$ and vice versa.

(CHANGE *DATUM* *FORM*) [Change Word]

This is the most flexible of all change words, since it enables the user to provide an arbitrary form describing what the new value should be, but it still highlights the fact that structure modification is to occur, and still enables the datum expression to appear only once. CHANGE sets *DATUM* to the value of *FORM**, where *FORM** is constructed from *FORM* by substituting the datum expression for every occurrence of the litatom DATUM. For example, (CHANGE (CAR X) (ITIMES DATUM 5)) translates as (CAR (RPLACA X (ITIMES (CAR X) 5))).

CHANGE is useful for expressing modifications that are not built-in and are not sufficiently common to justify defining a user-changeword. As for other changeword expressions, the user need not repeat the datum-expression and need not worry about multiple evaluation of the accessing form.

It is possible for the user to define new change words. To define a change word, say sub, that subtracts items from the current value of the datum, the user must put the property CLISPWORD, value (CHANGETRAN . sub) on both the upper and lower-case versions of sub:

```
(PUTPROP 'SUB 'CLISPWORD '(CHANGETRAN . sub))
(PUTPROP 'sub 'CLISPWORD '(CHANGETRAN . sub))
```

Then, the user must put (on the *lower*-case version of **sub** only) the property **CHANGEWORD**, with value *FN*. *FN* is a function that will be applied to a single argument, the whole **sub** form, and must return a form that Changetran can translate into an appropriate expression. This form should be a list structure with the atom **DATUM** used whenever the user wants an accessing expression for the current value of the datum to appear. The form (DATUM← *FORM*) (note that DATUM← is a single atom) should occur once in the expression; this specifies that an appropriate storing expression into the datum should occur at that point. For example, sub could be defined with:

```
(PUTPROP 'sub 'CHANGEWORD
         '(LAMBDA (FORM)
            (LIST 'DATUM←
                  (LIST 'IDIFFERENCE
                        'DATUM
                        (CONS 'IPLUS (CDDR FORM))))))
```

If the expression (sub (CAR X) A B) were encountered, the arguments to SUB would first be dwimified, and then the CHANGEWORD function would be passed the list (sub (CAR X) A B), and return (DATUM← (IDIFFERENCE DATUM (IPLUS A B))), which Changetran would convert to (CAR (RPLACA X (IDIFFERENCE (CAR X) (IPLUS A B)))).

Note: The **sub** changeword as defined above will always use IDIFFERENCE and IPLUS; **add** uses the correct addition operation depending on the current CLISP declarations.

3.9 USER DEFINED DATA TYPES

Note: The most convenient way to define new user data types is via DATATYPE record declarations (see page 3.7).

In addition to built-in data-types such as atoms, lists, arrays, etc., Interlisp provides a way of defining completely *new* classes of objects, with a fixed number of fields determined by the definition of the data type. Facilities are provided for declaring the name and *type* of the fields for a given class, creating objects of a given class, accessing and replacing the contents of each of the fields of such an object, as well as interrogating such objects.

In order to define a new class of objects, the user must supply a name for the new data type and specifications for each of its fields. Each field may contain either a pointer (i.e., any arbitrary Interlisp datum), an integer, a floating point number, or an *N*-bit integer. This is done via the function DECLAREDATATYPE:

(DECLAREDATATYPE *TYPENAME FIELDSPECS*) [Function]

TYPENAME is a literal atom, which specifies the name of the data type. *FIELDSPECS* is a list of "field specifications". Each field specification may be one of the following:

POINTER Field may contain any Interlisp datum.

FIXP Field contains an integer.

FLOATP Field contains a floating point number.

(BITS *N*) Field contains a non-negative integer less than 2^N.

DECLAREDATATYPE returns a list of "field descriptors", one for each element of *FIELDSPECS*. A field descriptor contains information about where within the datum the field is actually stored.

If *TYPENAME* is already declared a datatype, it is re-declared. If *FIELDSPECS* is NIL, *TYPENAME* is "undeclared".

(FETCHFIELD *DESCRIPTOR DATUM*) [Function]

Returns the contents of the field described by *DESCRIPTOR* from *DATUM*. *DESCRIPTOR* must be a "field descriptor" as returned by DECLAREDATATYPE. If *DATUM* is not an instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error DATUM OF INCORRECT TYPE.

In Interlisp-10, if *DESCRIPTOR* is quoted, FETCHFIELD compiles open. This capability is used by the record package.

(REPLACEFIELD *DESCRIPTOR DATUM NEWVALUE*) [Function]

Store *NEWVALUE* into the field of *DATUM* described by *DESCRIPTOR*. *DESCRIPTOR* must be a field descriptor as returned by DECLAREDATATYPE. If *DATUM* is not an instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error DATUM OF INCORRECT TYPE. Value is *NEWVALUE*.

(NCREATE *TYPENAME FROM*) [Function]

Creates and returns a new instance of datatype *TYPENAME*.

If *FROM* is also a datum of datatype *TYPENAME*, the fields of the new object are initialized to the values of the corresponding fields in *FROM*.

NCREATE will not work for built-in datatypes, such as ARRAYP, STRINGP, etc. If *TYPENAME* is not the type name of a previously declared *user* data type, generates an error, ILLEGAL DATA TYPE.

(GETFIELDSPECS *TYPENAME*) [Function]

Returns a list which is EQUAL to the *FIELDSPECS* argument given to DECLAREDATATYPE for *TYPENAME*; if *TYPENAME* is not a currently declared data-type, returns NIL.

(GETDESCRIPTORS *TYPENAME*) [Function]

Returns a list of field descriptors, EQUAL to the *value* of DECLAREDATATYPE for *TYPENAME*.

(USERDATATYPES) [Function]

Returns list of names of currently declared user data types.

Note that the user can define how user data types are to be printed via DEFPRINT (page 6.23), how they are to be evaluated by the interpreter via DEFEVAL (page 5.11), and how they are to be compiled by the compiler via COMPILETYPELST (page 12.9).

The DATATYPE facility in Interlisp-D is an extension of that found in Interlisp-10. Interlisp-D also accepts BYTE, WORD, and SIGNEDWORD as datatype field descriptors equivalent to BITS 8, BITS 16, and BETWEEN -2^{15} and 2^{15}-1 respectively. Interlisp-D will not move fields around in a user declaration if they pack into words and pointers as specified. POINTER fields take 24 bits and must be 32-bit right-justified.

CHAPTER 4

CONDITIONALS AND ITERATIVE STATEMENTS

In order to do any but the simplest computations, it is necessary to test values and execute expressions conditionally, and to execute expressions repeatedly. Interlisp supplies a large number of useful conditional and iterative constructs.

(COND *CLAUSE*$_1$ *CLAUSE*$_2$ ··· *CLAUSE*$_K$) [NLambda NoSpread Function]

The conditional function of Interlisp, COND, takes an indefinite number of arguments, called clauses. Each *CLAUSE*$_i$ is a list of the form (P_i C_{i1} ··· C_{iN}), where P_i is the predicate, and C_{i1} ··· C_{iN} are the consequents. The operation of COND can be paraphrased as:

IF P_1 THEN C_{11} ··· C_{1N} ELSEIF P_2 THEN C_{21} ··· C_{2N} ELSEIF P_3 ···

The clauses are considered in sequence as follows: the predicate P_1 of the clause *CLAUSE*$_i$ is evaluated. If the value of P_1 is "true" (non-NIL), the consequents C_{i1} ··· C_{iN} are evaluated in order, and the value of the COND is the value of C_{iN}, the last expression in the clause. If P_1 is "false" (EQ to NIL), then the remainder of *CLAUSE*$_i$ is ignored, and the next clause, *CLAUSE*$_{i+1}$, is considered. If no P_i is true for *any* clause, the value of the COND is NIL.

Note: If a clause has no consequents, and has the form (P_i), then if P_i evaluates to non-NIL, it is returned as the value of the COND. It is only evaluated once.

Example:

```
← (DEFINEQ (DOUBLE (X)
             (COND ((NUMBERP X) (PLUS X X))
                   ((STRINGP X) (CONCAT X X))
                   ((ATOM X) (PACK* X X))
                   (T (PRINT "unknown") X)
                   ((HORRIBLE-ERROR))]
(DOUBLE)
← (DOUBLE 5)
10
← (DOUBLE "FOO")
"FOOFOO"
← (DOUBLE 'BAR)
BARBAR
← (DOUBLE '(A B C))
"unknown"
(A B C)
```

A few points about this example: Notice that 5 is both a number and an atom, but it is "caught" by the NUMBERP clause before the ATOM clause. Also notice the predicate T, which is always true. This is the normal way to indicate a COND clause which will always be executed (if none of the preceeding clauses are true). (HORRIBLE-ERROR) will never be executed.

Note: The IF statement (page 4.4) provides an easier and more readable way of coding conditional expressions than COND.

(AND X_1 X_2 ··· X_N) [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument evaluates to NIL, AND immediately returns NIL (without evaluating the remaining arguments). If all of the arguments evaluate to non-NIL, the value of the last argument is returned. (AND) => T.

(OR X_1 X_2 ··· X_N) [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument is non-NIL, the value of that argument is returned by OR (without evaluating the remaining arguments). If all of the arguments evaluate to NIL, NIL is returned. (OR) => NIL.

AND and OR can be used as simple logical connectives, but note that they may not evaluate all of their arguments. This makes a difference if the evaluation of some of the arguments causes side-effects. Another result of this implementation of AND and OR is that they can be used as simple conditional statements. For example: (AND (LISTP *X*) (CDR *X*)) returns the value of (CDR *X*) if *X* is a list cell, otherwise it returns NIL without evaluating (CDR *X*). In general, this use of AND and OR should be avoided in favor of more explicit conditional statements in order to make programs more readable.

(SELECTQ *X* $CLAUSE_1$ $CLAUSE_2$ ··· $CLAUSE_K$ *DEFAULT*) [NLambda NoSpread Function]

Selects a form or sequence of forms based on the value of its first argument *X*. Each clause $CLAUSE_i$ is a list of the form (S_i C_{i1} ··· C_{iN}) where S_i is the selection key. The operation of SELECTQ can be paraphrased as:

IF $X = S_1$ THEN $C_{11} \cdots C_{1N}$ ELSEIF $X = S_2$ THEN ··· ELSE *DEFAULT*.

If S_i is an atom, the value of *X* is tested to see if it is EQ to S_i (which is not evaluated). If so, the expressions $C_{i1} \cdots C_{iN}$ are evaluated in sequence, and the value of the SELECTQ is the value of the last expression evaluated, i.e., C_{iN}.

If S_i is a list, the value of *X* is compared with each element (not evaluated) of S_i, and if *X* is EQ to any one of them, then $C_{i1} \cdots C_{iN}$ are evaluated as above.

If $CLAUSE_i$ is not selected in one of the two ways described, $CLAUSE_{i+1}$ is tested, etc., until all the clauses have been tested. If none is selected, *DEFAULT* is evaluated, and its value is returned as the value of the SELECTQ. *DEFAULT* must be present.

An example of the form of a SELECTQ is:

```
[SELECTQ MONTH
   (FEBRUARY (if (LEAPYEARP) then 29 else 28))
   ((APRIL JUNE SEPTEMBER NOVEMBER) 30)
   31]
```

If the value of MONTH is the litatom FEBRUARY, the SELECTQ returns 28 or 29 (depending on (LEAPYEARP)); otherwise if MONTH is APRIL, JUNE, SEPTEMBER, or NOVEMBER, the SELECTQ returns 30; otherwise it returns 31.

SELECTQ compiles open, and is therefore very fast; however, it will not work if the value of *x* is a list, a large integer, or floating point number, since SELECTQ uses EQ for all comparisons.

Note: The function SELCHARQ (page 2.13) is a version of SELECTQ that recognizes CHARCODE litatoms.

(SELECTC *X* *CLAUSE*$_1$ *CLAUSE*$_2$ ··· *CLAUSE*$_K$ *DEFAULT*) [NLambda NoSpread Function]

"SELECTQ-on-Constant." Similar to SELECTQ except that the selection keys are evaluated, and the result used as a SELECTQ-style selection key.

SELECTC is compiled as a SELECTQ, with the selection keys evaluated at compile-time. Therefore, the selection keys act like compile-time constants (see page 12.5). For example:

```
[SELECTC NUM
   ( (for X from 1 to 9 collect (TIMES X X)) "SQUARE" )
   "HIP"]
```

compiles as:

```
[SELECTQ NUM
   ( (1 4 9 16 25 36 49 64 81) "SQUARE" )
   "HIP"]
```

(PROG1 x_1 x_2 ··· x_N) [NLambda NoSpread Function]

Evaluates its arguments in order, and returns the value of its first argument x_1. For example, (PROG1 X (SETQ X Y)) sets X to Y, and returns X's original value.

(PROG2 x_1 x_2 ··· x_N) [Function]

Similar to PROG1. Evaluates its arguments in order, and returns the value of its second argument x_2.

(PROGN x_1 x_2 ··· x_N) [NLambda NoSpread Function]

PROGN evaluates each of its arguments in order, and returns the value of its last argument. PROGN is used to specify more than one computation where the syntax allows only one, e.g., (SELECTQ ··· (PROGN ···)) allows evaluation of several expressions as the default condition for a SELECTQ.

(PROG *VARLST* E_1 E_2 ··· E_N) [NLambda NoSpread Function]

This function allows the user to write an ALGOL-like program containing Interlisp expressions (forms) to be executed. The first argument, *VARLST*, is a list of local variables (must be NIL if no variables are used). Each atom in *VARLST* is treated as the name of a local variable and bound to NIL. *VARLST* can also contain lists of the form (atom form). In this case, atom is the name of the variable and is bound to the value of form. The evaluation takes place before any of the bindings are performed, e.g., (PROG ((X Y) (Y X)) ···) will bind local variable X to the value of Y (evaluated *outside* the PROG) and local variable Y to the value of X (outside the PROG). An attempt to use anything other than a literal atom as a PROG variable will cause an error, ARG NOT LITATOM. An attempt to use NIL or T as a PROG variable will cause an error, ATTEMPT TO BIND NIL OR T.

The rest of the PROG is a sequence of non-atomic statements (forms) and litatoms (labels). The forms are evaluated sequentially; the labels serve only as markers. The two special functions GO and RETURN alter this flow of control as described below. The value of the PROG is usually specified by the function RETURN. If no RETURN is executed before the PROG "falls off the end," the value of the PROG is NIL.

(GO *x*) [NLambda NoSpread Function]

GO is used to cause a transfer in a PROG. (GO L) will cause the PROG to evaluate forms starting at the label L (GO does not evaluate its argument). A GO can be used at any level in a PROG. If the label is not found, GO will search higher progs *within the same function*, e.g., (PROG ··· A ··· (PROG ··· (GO A))). If the label is not found in the function in which the PROG appears, an error is generated, UNDEFINED OR ILLEGAL GO.

(RETURN *x*) [Function]

A RETURN is the normal exit for a PROG. Its argument is evaluated and is immediately returned the value of the PROG in which it appears.

Note: If a GO or RETURN is executed in an interpreted function which is not a PROG, the GO or RETURN will be executed in the last interpreted PROG entered if any, otherwise cause an error.

GO or RETURN inside of a compiled function that is not a PROG is not allowed, and will cause an error at compile time.

As a corollary, GO or RETURN in a functional argument, e.g., to SORT, will not work compiled. Also, since NLSETQ's and ERSETQ's compile as *separate* functions, a GO or RETURN *cannot* be used inside of a compiled NLSETQ or ERSETQ if the corresponding PROG is outside, i.e., above, the NLSETQ or ERSETQ.

4.1 THE IF STATEMENT

The IF statement provides a way of way of specifying conditional expressions that is much easier and readable than using the COND function directly. CLISP translates expressions employing IF, THEN, ELSEIF, or ELSE into equivalent COND expressions. In general, statements of the form:

```
(IF AAA THEN BBB ELSEIF CCC THEN DDD ELSE EEE)
```

are translated to:

```
(COND (AAA BBB)
      (CCC DDD)
      (T EEE) )
```

The segment between IF or ELSEIF and the next THEN corresponds to the predicate of a COND clause, and the segment between THEN and the next ELSE or ELSEIF as the consequent(s). ELSE is the same as ELSEIF T THEN. These words are spelling corrected using the spelling list CLISPIFWORDSPLST. Lower case versions (if, then, elseif, else) may also be used.

If there is nothing following a THEN, or THEN is omitted entirely, then the resulting COND clause has a predicate but no consequent. For example, (IF X THEN ELSEIF ···) and (IF X ELSEIF ···) both translate to (COND (X) ···), which means that if X is not NIL, it is returned as the value of the COND.

CLISP considers IF, THEN, ELSE, and ELSEIF to have lower precedence than all infix and prefix operators, as well as Interlisp forms, so it is sometimes possible to omit parentheses around predicate or consequent forms. For example, (IF FOO X Y THEN ···) is equivalent to (IF (FOO X Y) THEN ···), and (IF X THEN FOO X Y ELSE ···) as equivalent to (IF X THEN (FOO X Y) ELSE ···). Essentially, CLISP determines whether the segment between THEN and the next ELSE or ELSEIF corresponds to one form or several and acts accordingly, occasionally interacting with the user to resolve ambiguous cases. Note that if FOO is bound as a variable, (IF FOO THEN ···) is translated as (COND (FOO ···)), so if a call to the *function* FOO is desired, use (IF (FOO) THEN ···).

4.2 THE ITERATIVE STATEMENT

The iterative statement (i.s.) in its various forms permits the user to specify complicated iterative statements in a straightforward and visible manner. Rather than the user having to perform the mental transformations to an equivalent Interlisp form using PROG, MAPC, MAPCAR, etc., the system does it for him. The goal was to provide a robust and tolerant facility which could "make sense" out of a wide class of iterative statements. Accordingly, the user should not feel obliged to read and understand in detail the description of each operator given below in order to use iterative statements.

An iterative statement is a form consisting of a number of special words (known as i.s. operators or i.s.oprs), followed by operands. Many i.s.oprs (FOR, DO, WHILE, etc.) are similar to iterative statements in other programming languages; other i.s.oprs (COLLECT, JOIN, IN, etc.) specify useful operations in a Lisp environment. Lower case versions of i.s.oprs (do, collect, etc.) can also be used. Here are some examples of iterative statements:

```
← (for X from 1 to 5 do (PRINT 'FOO))
FOO
FOO
FOO
FOO
FOO
NIL
← (for X from 2 to 10 by 2 collect (TIMES X X))
(4 16 36 64 100)
← (for X in '(A B 1 C 6.5 NIL (45)) count (NUMBERP X))
2
```

Iterative statements are implemented through CLISP, which translates the form into the appropriate PROG, MAPCAR, etc. Iterative statement forms are translated using all CLISP declarations in effect (standard/fast/undoable/ etc.); see page 16.9. Misspelled i.s.oprs are recognized and corrected using the spelling list CLISPFORWORDSPLST. The order of appearance of operators is never important; CLISP scans the entire statement before it begins to construct the equivalent Interlisp form. New i.s.oprs can be defined as described on page 4.13.

If the user defines a function by the same name as an i.s.opr (WHILE, TO, etc.), the i.s.opr will no longer have the CLISP interpretation when it appears as CAR of a form, although it will continue to be treated as an i.s.opr if it appears in the interior of an iterative statement. To alert the user, a warning message is printed, e.g., (WHILE DEFINED, THEREFORE DISABLED IN CLISP).

4.2.1 I.s.types

The following i.s.oprs are examples of a certain kind of iterative statement operator called an i.s.type. The i.s.type specifies what is to be done at each iteration. Its operand is called the "body" of the iterative statement. Each iterative statement must have one and only one i.s.type.

DO *FORM* [I.S. Operator]

Specifies what is to be done at each iteration. DO with no other operator specifies an infinite loop. If some explicit or implicit terminating condition is specified, the value of the i.s. is NIL. Translates to MAPC or MAP whenever possible.

COLLECT *FORM* [I.S. Operator]

Specifies that the value of *FORM* at each iteration is to be collected in a list, which is returned as the value of the i.s. when it terminates. Translates to MAPCAR, MAPLIST or SUBSET whenever possible.

When COLLECT translates to a PROG (e.g., if UNTIL, WHILE, etc. appear in the i.s.), the translation employs an open TCONC using two pointers similar to that used by the compiler for compiling MAPCAR. To disable this translation, perform (CLDISABLE 'FCOLLECT).

JOIN *FORM* [I.S. Operator]

Similar to COLLECT, except that the values of *FORM* at each iteration are NCONCed. Translates to MAPCONC or MAPCON whenever possible. /NCONC, /MAPCONC, and /MAPCON are used when the CLISP declaration UNDOABLE is in effect.

SUM *FORM* [I.S. Operator]

Specifies that the values of *FORM* at each iteration be added together and returned as the value of the i.s., e.g., (FOR I FROM 1 TO 5 SUM I↑2) is equal to 1+4+9+16+25. IPLUS, FPLUS, or PLUS will be used in the translation depending on the CLISP declarations in effect.

COUNT *FORM* [I.S. Operator]

Counts the number of times that *FORM* is true, and returns that count as its value.

ALWAYS *FORM* [I.S. Operator]

Returns T if the value of *FORM* is non-NIL for all iterations. (Note: returns NIL as soon as the value of *FORM* is NIL).

NEVER *FORM* [I.S. Operator]

Similar to ALWAYS, except returns T if the value of *FORM* is *never* true. (Note: returns NIL as soon as the value of *FORM* is non-NIL).

The following i.s.types explicitly refer to the iteration variable (i.v.) of the iterative statement, which is a variable set at each iteration. This is explained below under FOR.

THEREIS *FORM* [I.S. Operator]

Returns the first value of the i.v. for which *FORM* is non-NIL, e.g., (FOR X IN Y

THEREIS (NUMBERP X)) returns the first number in Y. (Note: returns the value of the i.v. as soon as the value of *FORM* is non-NIL).

LARGEST *FORM* [I.S. Operator]
SMALLEST *FORM* [I.S. Operator]

Returns the value of the i.v. that provides the largest/smallest value of *FORM*. $$EXTREME is always bound to the current greatest/smallest value, $$VAL to the value of the i.v. from which it came.

4.2.2 Iteration Variable I.s.oprs

FOR *VAR* [I.S. Operator]

Specifies the iteration variable (i.v.) which is used in conjunction with IN, ON, FROM, TO, and BY. The variable is rebound within the i.s., so the value of the variable outside the i.s. is not effected. Example:

```
← (SETQ X 55)
55
← (for X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
← X
55
```

FOR *VARS* [I.S. Operator]

VARS a list of variables, e.g., (FOR (X Y Z) IN ···). The first variable is the i.v., the rest are dummy variables. See BIND below.

FOR OLD *VAR* [I.S. Operator]

Similar to FOR, except that *VAR* is *not* rebound within the i.s., so the value of the i.v. outside of the i.s. is changed. Example:

```
← (SETQ X 55)
55
← (for old X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
← X
6
```

BIND *VAR* [I.S. Operator]
BIND *VARS* [I.S. Operator]

Used to specify dummy variables, which are bound locally within the i.s.

Note: FOR, FOR OLD, and BIND variables can be initialized by using the form *VAR*←*FORM*:

(FOR OLD (X←*FORM*) BIND (Y←*FORM*) ···)

IN *FORM* [I.S. Operator]

Specifies that the i.s. is to iterate down a list with the i.v. being reset to the corresponding element at each iteration. For example, (FOR X IN Y DO ···) corresponds to (MAPC Y (FUNCTION (LAMBDA (X) ···))). If no i.v. has been specified, a dummy is supplied, e.g., (IN Y COLLECT CADR) is equivalent to (MAPCAR Y (FUNCTION CADR)).

ON *FORM* [I.S. Operator]

Same as IN except that the i.v. is reset to the corresponding *tail* at each iteration. Thus IN corresponds to MAPC, MAPCAR, and MAPCONC, while ON corresponds to MAP, MAPLIST, and MAPCON.

Note: for both IN and ON, *FORM* is evaluated before the main part of the i.s. is entered, i.e. *outside* of the scope of any of the bound variables of the i.s. For example, (FOR X BIND (Y←'(1 2 3)) IN Y ···) will map down the list which is the value of Y evaluated *outside* of the i.s., *not* (1 2 3).

IN OLD *VAR* [I.S. Operator]

Specifies that the i.s. is to iterate down *VAR*, with *VAR* itself being reset to the corresponding tail at each iteration, e.g., after (FOR X IN OLD L DO ··· UNTIL ···) finishes, L will be some tail of its original value.

IN OLD (*VAR←FORM*) [I.S. Operator]

Same as IN OLD *VAR*, except *VAR* is first set to value of *FORM*.

ON OLD *VAR* [I.S. Operator]

Same as IN OLD *VAR* except the i.v. is reset to the current value of *VAR* at each iteration, instead of to (CAR *VAR*).

ON OLD (*VAR←FORM*) [I.S. Operator]

Same as ON OLD *VAR*, except *VAR* is first set to value of *FORM*.

INSIDE *FORM* [I.S. Operator]

Similar to IN, except treats first non-list, non-NIL tail as the last element of the iteration, e.g., INSIDE '(A B C D . E) iterates five times with the i.v. set to E on the last iteration. INSIDE 'A is equivalent to INSIDE '(A), which will iterate once.

FROM *FORM* [I.S. Operator]

Used to specify an initial value for a numerical i.v. The i.v. is automatically incremented by 1 after each iteration (unless BY is specified). If no i.v. has been specified, a dummy i.v. is supplied and initialized, e.g., (FROM 2 TO 5 COLLECT SQRT) returns (1.414 1.732 2.0 2.236).

TO *FORM* [I.S. Operator]

Used to specify the final value for a numerical i.v. If FROM is not specified, the i.v. is initialized to 1. If no i.v. has been specified, a dummy i.v. is supplied and initialized. If BY is not specified, the i.v. is automatically incremented by 1 after each iteration.[1] When the i.v. is definitely being *incremented*, i.e., either BY is not specified, or its operand is a positive number, the i.s. terminates when the i.v. exceeds the value of *FORM* e.g., (FOR X FROM 1 TO 10 --) is equivalent to (FOR X FROM 1 UNTIL (X GT 10) --). Similarly, when the i.v. is definitely

[1] except when both the operands to TO and FROM are numbers, and TO's operand is less than FROM's operand, e.g., FROM 10 TO 1, in which case the i.v. is decremented by 1 after each iteration. In this case, the i.s. terminates when the i.v. becomes *less* than the value of *FORM*.

being decremented the i.s. terminates when the i.v. becomes *less* than the value of *FORM* (see description of BY).

Note: *FORM* is evaluated only once, when the i.s. is first entered, and its value bound to a temporary variable against which the i.v. is checked each interation. If the user wishes to specify an i.s. in which the value of the boundary condition is recomputed each iteration, he should use WHILE or UNTIL instead of TO.

BY *FORM* (with IN/ON) [I.S. Operator]

If IN or ON have been specified, the value of *FORM* determines the *tail* for the next iteration, which in turn determines the value for the i.v. as described earlier, i.e., the new i.v. is CAR of the tail for IN, the tail itself for ON. In conjunction with IN, the user can refer to the current tail within *FORM* by using the i.v. or the operand for IN/ON, e.g., (FOR Z IN L BY (CDDR *Z*) ···) or (FOR Z IN L BY (CDDR *L*) ···). At translation time, the name of the internal variable which holds the value of the current tail is substituted for the i.v. throughout *FORM*. For example, (FOR X IN Y BY (CDR (MEMB 'FOO (CDR X))) COLLECT X) specifies that after each iteration, CDR of the current tail is to be searched for the atom FOO, and (CDR of) this latter tail to be used for the next iteration.

BY *FORM* (without IN/ON) [I.S. Operator]

If IN or ON have not been used, BY specifies how the i.v. itself is reset at each iteration. If FROM or TO have been specified, the i.v. is known to be numerical, so the new i.v. is computed by adding the value of *FORM* (which is reevaluated each iteration) to the current value of the i.v., e.g., (FOR N FROM 1 TO 10 BY 2 COLLECT N) makes a list of the first five odd numbers.

If *FORM* is a positive number,[2] the i.s. terminates when the value of the i.v. *exceeds* the value of TO's operand. If *FORM* is a negative number, the i.s. terminates when the value of the i.v. becomes *less* than TO's operand, e.g., (FOR I FROM N TO M BY -2 UNTIL (I LT M) ...). Otherwise, the terminating condition for each iteration depends on the value of *FORM* for that iteration: if *FORM*<0, the test is whether the i.v. is less than TO's operand, if *FORM*>0 the test is whether the i.v. exceeds TO's operand, otherwise if *FORM*=0, the i.s. terminates unconditionally.

If FROM or TO have not been specified and *FORM* is not a number, the i.v. is simply reset to the value of *FORM* after each iteration, e.g., (FOR I FROM N BY M ...) is equivalent to (FOR I←N BY (IPLUS I M) ...).

AS *VAR* [I.S. Operator]

Used to specify an iterative statement involving more than one iterative variable, e.g., (FOR X IN Y AS U IN V DO --) corresponds to MAP2C. The i.s. terminates when any of the terminating conditions are met, e.g., (FOR X IN Y AS I FROM 1 TO 10 COLLECT X) makes a list of the first ten elements of Y, or however many elements there are on Y if less than 10.

The operand to AS, *VAR*, specifies the new i.v. For the remainder of the i.s., or until another AS is encountered, all operators refer to the new i.v. For

[2] *FORM* itself, not its value, which in general CLISP would have no way of knowing in advance.

example, (FOR I FROM 1 TO N1 AS J FROM 1 TO N2 BY 2 AS K FROM N3 TO 1 BY -1 --) terminates when I exceeds N1, or J exceeds N2, or K becomes less than 1. After each iteration, I is incremented by 1, J by 2, and K by -1.

OUTOF *FORM* [I.S. Operator]

For use with generators (page 7.13). On each iteration, the i.v. is set to successive values returned by the generator. The i.s. terminates when the generator runs out.

4.2.3 Condition I.s.oprs

WHEN *FORM* [I.S. Operator]

Provides a way of excepting certain iterations. For example, (FOR X IN Y COLLECT X WHEN (NUMBERP X)) collects only the elements of Y that are numbers.

UNLESS *FORM* [I.S. Operator]

Same as WHEN except for the difference in sign, i.e., WHEN Z is the same as UNLESS (NOT Z).

WHILE *FORM* [I.S. Operator]

Provides a way of terminating the i.s. WHILE *FORM* evaluates *FORM* *before* each iteration, and if the value is NIL, exits.

UNTIL *FORM* [I.S. Operator]

Same as WHILE except for difference in sign, i.e., WHILE X is equivalent to UNTIL (NOT X).

UNTIL *N* (*N* a number) [I.S. Operator]

Equivalent to UNTIL (*I.V.* GT *N*).

REPEATWHILE *FORM* [I.S. Operator]

Same as WHILE except the test is performed after the evalution of the body, but before the i.v. is reset for the next iteration.

REPEATUNTIL *FORM* [I.S. Operator]

Same as UNTIL, except the test is performed after the evaluation of the body.

REPEATUNTIL *N* (*N* a number) [I.S. Operator]

Equivalent to REPEATUNTIL (*I.V.* GT *N*).

4.2.4 Other I.s.oprs

FIRST *FORM* [I.S. Operator]

FORM is evaluated once before the first iteration, e.g., (FOR X Y Z IN L FIRST (FOO Y Z) ···), and FOO could be used to initialize Y and Z.

FINALLY *FORM* [I.S. Operator]

FORM is evaluated after the i.s. terminates. For example, (FOR X IN

L BIND Y←0 DO (IF ATOM X THEN Y←Y+1) FINALLY (RETURN Y)) will return the number of atoms in L.

EACHTIME *FORM* [I.S. Operator]

FORM is evaluated at the beginning of each iteration before, and regardless of, any testing. For example, consider,

```
(FOR I FROM 1 TO N
    DO (··· (FOO I) ···)
    UNLESS (··· (FOO I) ···)
    UNTIL (··· (FOO I) ···))
```

The user might want to set a temporary variable to the value of (FOO I) in order to avoid computing it three times each iteration. However, without knowing the translation, he would not know whether to put the assignment in the operand to DO, UNLESS, or UNTIL, i.e., which one would be executed first. He can avoid this problem by simply writing EACHTIME (SETQ J (FOO I)).

DECLARE: *DECL* [I.S. Operator]

Inserts the form (DECLARE *DECL*) immediately following the PROG variable list in the translation, or, in the case that the translation is a mapping function rather than a PROG, immediately following the argument list of the lambda expression in the translation. This can be used to declare variables bound in the iterative statement to be compiled as local or special variables (see page 12.4). For example (FOR X IN Y DECLARE: (LOCALVARS X) ···). Several DECLARE:s can apppear in the same i.s.; the declarations are inserted in the order they appear.

DECLARE *DECL* [I.S. Operator]

Same as DECLARE:.

Note that since DECLARE is also the name of a function, DECLARE cannot be used as an i.s. operator when it appears as CAR of a form, i.e. as the first i.s. operator in an iterative statement. However, declare (lower-case version) *can* be the first i.s. operator.

ORIGINAL *I.S.OPR OPERAND* [I.S. Operator]

I.S.OPR will be translated using its original, built-in interpretation, independent of any user defined i.s. operators. See page 4.13.

There are also a number of i.s.oprs that make it easier to create iterative statements that use the clock, looping for a given period of time. See Timers, page 14.11.

4.2.5 Miscellaneous

- Lowercase versions of all i.s. operators are equivalent to the uppercase, e.g., (for X in Y ···).

- Each i.s. operator is of lower precedence than all Interlisp forms, so parentheses around the operands can be omitted, and will be supplied where necessary, e.g., BIND (X Y Z) can be written BIND X Y Z, OLD (X←*FORM*) as OLD X←*FORM*, WHEN (NUMBERP X) as WHEN NUMBERP X, etc.

- RETURN or GO may be used in any operand. (In this case, the translation of the iterative statement will always be in the form of a PROG, never a mapping function.) RETURN means return from the i.s. (with the indicated value), *not* from the function in which the i.s appears. GO refers to a label elsewhere in the function in which the i.s. appears, except for the labels **$$LP**, **$$ITERATE**, and **$$OUT** which are reserved, as described below.

• In the case of FIRST, FINALLY, EACHTIME, DECLARE: or one of the i.s.types, e.g., DO, COLLECT, SUM, etc., the operand can consist of more than one form, e.g., COLLECT (PRINT X:1) X:2, in which case a PROGN is supplied.

• Each operand can be the name of a function, in which case it is applied to the (last) i.v., e.g., (FOR X IN Y DO PRINT WHEN NUMBERP) is the same as (FOR X IN Y DO (PRINT X) WHEN (NUMBERP X)). Note that the i.v. need not be explicitly specified, e.g., (IN Y DO PRINT WHEN NUMBERP) will work.

For i.s.types, e.g., DO, COLLECT, JOIN, the function is always applied to the first i.v. in the i.s., whether explicity named or not. For example, (IN Y AS I FROM 1 TO 10 DO PRINT) prints elements on Y, not integers between 1 and 10.

Note that this feature does not make much sense for FOR, OLD, BIND, IN, or ON, since they "operate" before the loop starts, when the i.v. may not even be bound.

In the case of BY in conjunction with IN, the function is applied to the current *tail* e.g., FOR X IN Y BY CDDR ... is the same as FOR X IN Y BY (CDDR X)....

• While the exact form of the translation of an iterative statement depends on which operators are present, a PROG will always be used whenever the i.s. specifies dummy variables, i.e., if a BIND operator appears, or there is more than one variable specified by a FOR operator, or a GO, RETURN, or a reference to the variable $$VAL appears in any of the operands. When a PROG is used, the form of the translation is:

```
(PROG VARIABLES
      {initialize}
$$LP  {eachtime}
      {test}
      {body}
$$ITERATE
      {aftertest}
      {update}
      (GO $$LP)
$$OUT {finalize}
      (RETURN $$VAL))
```

where {test} corresponds to that portion of the loop that tests for termination and also for those iterations for which {body} is not going to be executed, (as indicated by a WHEN or UNLESS); {body} corresponds to the operand of the i.s.type, e.g., DO, COLLECT, etc.; {aftertest} corresponds to those tests for termination specified by REPEATWHILE or REPEATUNTIL; and {update} corresponds to that part that resets the tail, increments the counter, etc. in preparation for the next iteration. {initialize}, {finalize}, and {eachtime} correspond to the operands of FIRST, FINALLY, and EACHTIME, if any.

Note that since {body} always appears at the top level of the PROG, the user can insert labels in {body}, and GO to them from within {body} or from other i.s. operands, e.g., (FOR X IN Y FIRST (GO A) DO (FOO) A (FIE)). However, since {body} is dwimified as a list of forms, the label(s) should be added to the dummy variables for the iterative statement in order to prevent their being dwimified and possibly "corrected", e.g., (FOR X IN Y BIND A FIRST (GO A) DO (FOO) A (FIE)). The user can also GO to $$LP, $$ITERATE, or $$OUT, or explicitly set $$VAL.

4.2.6 Errors in Iterative Statements

An error will be generated and an appropriate diagnostic printed if any of the following conditions hold:

1. Operator with null operand, i.e., two adjacent operators, as in FOR X IN Y UNTIL DO --

2. Operand consisting of more than one form (except as operand to FIRST, FINALLY, or one of the i.s.types), e.g., FOR X IN Y (PRINT X) COLLECT --.

3. IN, ON, FROM, TO, or BY appear twice in same i.s.

4. Both IN and ON used on same i.v.

5. FROM or TO used with IN or ON on same i.v.

6. More than one i.s.type, e.g., a DO and a SUM.

In 3, 4, or 5, an error is not generated if an intervening AS occurs.

If an error occurs, the i.s. is left unchanged.

If no DO, COLLECT, JOIN or any of the other i.s.types are specified, CLISP will first attempt to find an operand consisting of more than one form, e.g., FOR X IN Y *(PRINT X)* WHEN ATOM X, and in this case will insert a DO after the first form. (In this case, condition 2 is not considered to be met, and an error is not generated.) If CLISP cannot find such an operand, and no WHILE or UNTIL appears in the i.s., a warning message is printed: NO DO, COLLECT, OR JOIN: followed by the i.s.

Similarly, if no terminating condition is detected, i.e., no IN, ON, WHILE, UNTIL, TO, or a RETURN or GO, a warning message is printed:[3] POSSIBLE NON-TERMINATING ITERATIVE STATEMENT: followed by the iterative statement. However, since the user may be planning to terminate the i.s. via an error, control-E, or a RETFROM from a lower function, the i.s. is still translated.

4.2.7 Defining New Iterative Statement Operators

The following function is available for defining new or redefining existing iterative statement operators:

(I.S.OPR *NAME FORM OTHERS EVALFLG*) [Function]

NAME is the name of the new i.s.opr. If *FORM* is a list, *NAME* will be a new *i.s.type* (see page 4.6), and *FORM* its body.

OTHERS is an (optional) list of additional i.s. operators and operands which will be added to the i.s. at the place where *NAME* appears. If *FORM* is NIL, *NAME* is a new i.s.opr defined entirely by *OTHERS*.

[3] unless the value of CLISPI.S.GAG is T (initially NIL).

In both *FORM* and *OTHERS*, the atom $$VAL can be used to reference the value to be returned by the i.s., I.V. to reference the current i.v., and BODY to reference *NAME*'s operand. In other words, the current i.v. will be substituted for all instances of I.V. and *NAME*'s operand will be substituted for all instances of BODY throughout *FORM* and *OTHERS*.

If *EVALFLG* is T, *FORM* and *OTHERS* are evaluated at translation time, and their values used as described above. LSTVARS is a list of dummy variable names used by the iterative statement translator. If the user wishes to obtain a dummy variable for use in translation, and be sure it does not clash with a dummy variable already used by some other i.s. operators, he can use CAR of LSTVARS, and reset LSTVARS to (CDR LSTVARS).

If *NAME* was previously an i.s.opr and is being redefined, the message (*NAME* REDEFINED) will be printed (unless DFNFLG=T), and all expressions using the i.s.opr *NAME* that have been translated will have their translations discarded.

For example, for COLLECT, *FORM* would be (SETQ $$VAL (NCONC1 $$VAL BODY)).

For SUM, *FORM* would be ($$VAL←$$VAL+BODY),[4] *OTHERS* would be (FIRST $$VAL←0).

For NEVER, *FORM* would be (IF BODY THEN $$VAL←NIL (GO $$OUT))).[5]

For THEREIS, *FORM* would be (IF BODY THEN $$VAL←I.V. (GO $$OUT)).

Examples:

To define RCOLLECT, a version of COLLECT which uses CONS instead of NCONC1 and then reverses the list of values:

```
(I.S.OPR 'RCOLLECT
         '($$VAL←(CONS BODY $$VAL))
         '(FINALLY (RETURN (DREVERSE $$VAL)))]
```

To define TCOLLECT, a version of COLLECT which uses TCONC:

```
(I.S.OPR 'TCOLLECT
         '(TCONC $$VAL BODY)
         '(FIRST $$VAL←(CONS) FINALLY (RETURN (CAR $$VAL)))]
```

To define PRODUCT:

```
(I.S.OPR 'PRODUCT
         '($$VAL←$$VAL*BODY)
         '(FIRST $$VAL←1)]
```

To define UPTO, a version of TO whose operand is evaluated only once:

[4] $$VAL+BODY is used instead of (IPLUS $$VAL BODY) so that the choice of function used in the translation, i.e., IPLUS, FPLUS, or PLUS, will be determined by the declarations then in effect.

[5] (IF BODY THEN RETURN NIL) would exit from the i.s. immediately and therefore not execute the operations specified via a FINALLY (if any).

```
(I.S.OPR 'UPTO
         NIL
         '(BIND $$FOO←BODY TO $$FOO)]
```

To redefine TO so that instead of recomputing *FORM* each iteration, a variable is bound to the value of *FORM*, and then that variable is used:

```
(I.S.OPR 'TO
         NIL
         '(BIND $$END FIRST $$END←BODY ORIGINAL TO $$END)]
```

Note the use of ORIGINAL to redefine TO in terms of its original definition. ORIGINAL is intended for use in redefining built-in operators, since their definitions are not accessible, and hence not directly modifiable. Thus if the operator had been defined by the user via I.S.OPR, ORIGINAL would not obtain its original definition. In this case, one presumably would simply modify the i.s.opr definition.

I.S.OPR can also be used to define synonyms for already defined i.s. operators by calling I.S.OPR with *FORM* an atom, e.g., (I.S.OPR 'WHERE 'WHEN) makes WHERE be the same as WHEN. Similarly, following (I.S.OPR 'ISTHERE 'THEREIS), one can write (ISTHERE ATOM IN Y), and following (I.S.OPR 'FIND 'FOR) and (I.S.OPR 'SUCHTHAT 'THEREIS), one can write (FIND X IN Y SUCHTHAT X MEMBER Z). In the current system, WHERE is synonymous with WHEN, SUCHTHAT and ISTHERE with THEREIS, FIND with FOR, and THRU with TO.

If *FORM* is the atom MODIFIER, then *NAME* is defined as an i.s.opr which can immediately follow another i.s. operator (i.e., an error will not be generated, as described previously). *NAME* will not terminate the scope of the previous operator, and will be stripped off when DWIMIFY is called on its operand. OLD is an example of a MODIFIER type of operator. The MODIFIER feature allows the user to define i.s. operators similar to OLD, for use in conjunction with some other user defined i.s.opr which will produce the appropriate translation.

The file package command I.S.OPRS(page 11.25) will dump the definition of i.s.oprs. (I.S.OPRS PRODUCT UPTO) as a file package command will print suitable expressions so that these iterative statement operators will be (re)defined when the file is loaded.

CHAPTER 5

FUNCTION DEFINITION, MANIPULATION, AND EVALUATION

The Interlisp programming system is designed to help the user define and debug functions. Developing an applications program in Interlisp involves defining a number of functions in terms of the system primitives and other user-defined functions. Once defined, the user's functions may be referenced exactly like Interlisp primitive functions, so the programming process can be viewed as extending the Interlisp language to include the required functionality.

The user defines a function with a list expressions known as an EXPR. An EXPR specifies if the function has a fixed or variable number of arguments, whether these arguments are evaluated or not, the function argument names, and a series of forms which define the behavior of the function. For example:

```
(LAMBDA (X Y) (PRINT X) (PRINT Y))
```

A function defined with this EXPR would have two evaluated arguments, X and Y, and it would execute (PRINT X) and (PRINT Y) when evaluated. Other types of EXPRs are described below.

A function is defined by putting an EXPR in the function definition cell of a litatom. There are a number of functions for accessing and setting function definition cells, but one usually defines a function with DEFINEQ (page 5.9). For example:

```
← (DEFINEQ (FOO (LAMBDA (X Y) (PRINT X) (PRINT Y))
(FOO)
```

The expression above will define the function FOO to have the EXPR definition (LAMBDA (X Y) (PRINT X) (PRINT Y)). After being defined, this function may be evaluated just like any system function:

```
← (FOO 3 (IPLUS 3 4))
3
7
7
←
```

All function definition cells do not contain EXPRs. The compiler (page 12.1) translates EXPR definitions into compiled code objects, which execute much faster. In Interlisp-10, many primitive system functions are defined with machine code objects known as SUBRs. Interlisp provides a number of "function type functions" which determine how a given function is defined (EXPR/compiled code/SUBR), the number and names of function arguments, etc. See page 5.6.

Usually, functions are evaluated automatically when they appear within another function or when typed into Interlisp. However, sometimes it is useful to envoke the Interlisp interpreter explicitly to apply a given "functional argument" to some data. There are a number of functions which will apply a given function repeatedly. For example, MAPCAR will apply a function (or an EXPR) to all of the elements of a list, and return the values returned by the function:

```
← (MAPCAR '(1 2 3 4 5) '(LAMBDA (X) (ITIMES X X))
(1 4 9 16 25)
```

When using functional arguments, there are a number of problems which can arise, related with accessing free variables from within a function argument. Many times these problems can be solved using the function `FUNCTION` to create a `FUNARG` object (see page 5.15).

The macro facility provides another way of specifying the behavior of a function (see page 5.17). Macros are very useful when developing code which should run very quickly, which should be compiled differently than it is interpreted, or which should run differently in different implementations of Interlisp.

5.1 FUNCTION TYPES

Interlisp functions are defined using list expressions called `EXPR`s. An `EXPR` is a list of the form (*LAMBDA-WORD ARG-LIST* $FORM_1$ ··· $FORM_N$). *LAMBDA-WORD* determines whether the arguments to this function will be evaluated or not, *ARG-LIST* determines the number and names of arguments, and $FORM_1$ ··· $FORM_N$ are a series of forms to be evaluated after the arguments are bound to the local variables in *ARG-LIST*.

If *LAMBDA-WORD* is the litatom `LAMBDA`, then the arguments to the function are evaluated. If *LAMBDA-WORD* is the litatom `NLAMBDA`, then the arguments to the function are not evaluated. Functions which evaluate or don't evaluate their arguments are therefore known as "lambda" or "nlambda" functions, respectively.

If *ARG-LIST* is `NIL` or a list of litatoms, this indicates a function with a fixed number of arguments. Each litatom is the name of an argument for the function defined by this expression. The process of binding these litatoms to the individual arguments is called "spreading" the arguments, and the function is called a "spread" function. If the argument list is any litatom other than `NIL`, this indicates a function with a variable number of arguments, known as a "nospread" function.

If *ARG-LIST* is anything other than a litatom or a list of litatoms, such as `(LAMBDA "FOO" ···)`, attempting to use this `EXPR` will generate an `ARG NOT LITATOM` error. In addition, if `NIL` or `T` is used as an argument name, the error `ATTEMPT TO BIND NIL OR T` is generated.

These two parameters (lambda/nlambda and spread/nospread) may be specified independently, so there are four main function types, known as lambda-spread, nlambda-spread, lambda-nospread, and nlambda-nospread functions. Each one has a different form, and is used for a different purpose. These four function types are described more fully below.

Note: The Lambdatran lispusers package provides facilities for creating new function types which evaluate/spread their arguments in different ways than those provided by Interlisp. See page 23.16.

5.1.1 Lambda-Spread Functions

Lambda-spread functions take a fixed number of evaluated arguments. This is the most common function type. A lambda-spread `EXPR` has the form:

(`LAMBDA` (ARG_1 ··· ARG_M) $FORM_1$ ··· $FORM_N$)

The argument list $(\mathit{ARG}_1 \cdots \mathit{ARG}_M)$ is a list of litatoms that gives the number and names of the formal arguments to the function. If the argument list is () or NIL, this indicates that the function takes no arguments. When a lambda-spread function is applied to some arguments, the arguments are evaluated, and bound to the local variables $\mathit{ARG}_1 \cdots \mathit{ARG}_M$. Then, $\mathit{FORM}_1 \cdots \mathit{FORM}_N$ are evaluated in order, and the value of the function is the value of FORM_N.

```
← (DEFINEQ (FOO (LAMBDA (X Y) (PRINT X) (PRINT Y))) )
(FOO)
← (FOO 99 (PLUS 3 4))
99
7
7
←
```

In the above example, the function FOO defined by (LAMBDA (X Y) (PRINT X) (PRINT Y)) is applied to the arguments 99 and (PLUS 3 4), these arguments are evaluated (giving 99 and 7), the local variable X is bound to 99 and Y to 7, (PRINT X) is evaluated, printing 99, (PRINT Y) is evaluated, printing 7, and 7 (the *value* of (PRINT Y)) is returned as the value of the function.

A standard feature of the Interlisp system is that no error occurs if a spread function is called with too many or too few arguments. If a function is called with too many arguments, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as NIL. In fact, a spread function cannot distinguish between being given NIL as an argument, and not being given that argument, e.g., (FOO) and (FOO NIL) are *exactly* the same for spread functions. If it is necessary to distinguish between these two cases, use an nlambda function and explicitly evaluate the arguments with the EVAL function (page 5.11).

5.1.2 Nlambda-Spread Functions

Nlambda-spread functions take a fixed number of unevaluated arguments. An nlambda-spread EXPR has the form:

(NLAMBDA $(\mathit{ARG}_1 \cdots \mathit{ARG}_M)$ $\mathit{FORM}_1 \cdots \mathit{FORM}_N$)

Nlambda-spread functions are evaluated similarly to lambda-spread functions, except that the arguments are not evaluated before being bound to the variables $\mathit{ARG}_1 \cdots \mathit{ARG}_M$.

```
← (DEFINEQ (FOO (NLAMBDA (X Y) (PRINT X) (PRINT Y))) )
(FOO)
← (FOO 99 (PLUS 3 4))
99
(PLUS 3 4)
(PLUS 3 4)
←
```

In the above example, the function FOO defined by (NLAMBDA (X Y) (PRINT X) (PRINT Y)) is applied to the arguments 99 and (PLUS 3 4), these arguments are bound unevaluated to X and Y, (PRINT X) is evaluated, printing 99, (PRINT Y) is evaluated, printing (PLUS 3 4), and the list (PLUS 3 4) is returned as the value of the function.

Note: Functions can be defined so that all of their arguments are evaluated (lambda functions) or none are evaluated (nlambda functions). If it is desirable to write a function which only evaluates *some* of its arguments (e.g. SETQ), the function should be defined as an nlambda, with some arguments explicitly evaluated using the function EVAL (page 5.11). If this is done, the user should put the litatom EVAL on the property list of the function under the property INFO. This informs various system packages such as DWIM, CLISP, and Masterscope that this function in fact *does* evaluate its arguments, even though it is an nlambda.

5.1.3 Lambda-Nospread Functions

Lambda-nospread functions take a variable number of evaluated arguments. A lambda-nospread EXPR has the form:

(LAMBDA *VAR* *FORM*$_1$ ··· *FORM*$_N$)

VAR may be any litatom, except NIL and T. When a lambda-nospread function is applied to some arguments, each of these arguments is evaluated and the values stored on the pushdown list. *VAR* is then bound to the *number* of arguments which have been evaluated. For example, if FOO is defined by (LAMBDA X ···), when (FOO A B C) is evaluated, A, B, and C are evaluated and X is bound to 3. *VAR* should *never* be reset.

The following functions are used for accessing the arguments of lambda-nospread functions:

(ARG *VAR* *M*) [NLambda Function]

Returns the *M*th argument for the lambda-nospread function whose argument list is *VAR*. *VAR* is the *name* of the atomic argument list to a lambda-nospread function, and is not evaluated; *M* is the number of the desired argument, and is evaluated. The value of ARG is undefined for *M* less than or equal to 0 or greater than the *value* of *VAR*.

(SETARG *VAR* *M* *X*) [NLambda Function]

Sets the *M*th argument for the lambda-nospread function whose argument list is *VAR* to *X*. *VAR* is not evaluated; *M* and *X* are evaluated. *M* should be between 1 and the value of *VAR*.

In the example below, the function FOO is defined to print all of the evaluated arguments it is given, and return NIL (the value of the for statement).

```
← (DEFINEQ (FOO
             (LAMBDA X
                (for ARGNUM from 1 to X do (PRINT (ARG X ARGNUM))))) )
(FOO)
← (FOO 99 (PLUS 3 4))
99
7
NIL
← (FOO 99 (PLUS 3 4) (TIMES 3 4))
99
7
12
NIL
```

←

5.1.4 Nlambda-Nospread Functions

Nlambda-nospread functions take a variable number of unevaluated arguments. An nlambda-nospread EXPR has the form:

(NLAMBDA *VAR* *FORM*$_1$ ··· *FORM*$_N$)

VAR may be any litatom, except NIL and T. Though similar in form to lambda-nospread EXPRs, an nlambda-nospread is evaluated quite differently. When an nlambda-nospread function is applied to some arguments, *VAR* is simply bound to a list of the unevaluated arguments. The user may pick apart this list, and evaluate different arguments.

In the example below, FOO is defined to print (and then return) the reverse of list of arguments it is given (unevaluated):

```
← (DEFINEQ (FOO (NLAMBDA X (REVERSE X))))
(FOO)
← (FOO 99 (PLUS 3 4))
((PLUS 3 4) 99)
((PLUS 3 4) 99)
← (FOO 99 (PLUS 3 4) (TIMES 3 4))
((TIMES 3 4) (PLUS 3 4) 99)
((TIMES 3 4) (PLUS 3 4) 99)
←
```

5.1.5 Compiled Functions

Functions defined by EXPRs can be compiled by the Interlisp compiler (page 12.1), which produces compiled code objects, which execute more quickly than the corresponding EXPR code. Functions defined by compiled code objects may have the same four types as EXPRs (lambda/nolambda, spread/nospread). Functions created by the compiler are referred to as compiled functions.

5.1.6 SUBRs

In Interlisp-10, basic built-in functions such as CONS, CAR, and COND are handcoded in machine language. These functions are known as "SUBRs." Functions defined as SUBRs can be lambda/nolambda or spread/nospread, the same four function types as EXPR functions.

SUBRs are called in a special way, so their definitions are stored differently than those of compiled or interpreted functions. GETD of a SUBR returns a dotted pair, CAR of which is an encoding of the ARGTYPE and number of arguments of the SUBR, and CDR of which is the address of the first instruction. Note that each GETD of a subr performs a CONS. Similarly, PUTD of a definition of the form (*NUMBER* . *ADDRESS*), where *NUMBER* and *ADDRESS* are in the appropriate ranges, stores the definition as a SUBR.

5.1.7 Function Type Functions

There are a variety of functions used for examining the type, argument list, etc. of functions. These functions may be given either a litatom, in which case they obtain the function definition from the litatom's definition cell, or a function definition itself.

(FNTYP *FN*) [Function]

Returns NIL if *FN* is not a function definition or the name of a defined function. Otherwise FNTYP returns one of the following twelve litatoms:

	Expressions	Compiled	Built-In
Lambda-Spread	EXPR	CEXPR	SUBR
Nlambda-Spread	FEXPR	CFEXPR	FSUBR
Lambda-Nospread	EXPR*	CEXPR*	SUBR*
Nlambda-Nospread	FEXPR*	CFEXPR*	FSUBR*

The types in the first column are all defined by EXPRs. The types in the second column are compiled versions of the types in the first column, as indicated by the prefix C. In the third column are the parallel types for built-in subroutines (only in Interlisp-10). Functions of types in the first two rows have a fixed number of arguments, i.e., are spread functions. Functions in the third and fourth rows have an indefinite number of arguments, as indicated by the suffix *. The prefix F indicates unevaluated arguments. Thus, for example, a CFEXPR* is a compiled nospread-nlambda function.

FNTYP returns the litatom FUNARG if *FN* is a FUNARG expression. See page 5.15.

(EXPRP *FN*) [Function]

Returns T if (FNTYP *FN*) is either EXPR, FEXPR, EXPR*, or FEXPR*, i.e., first column of FNTYPs; NIL otherwise. However, (EXPRP *FN*) is also true if *FN* is (has) a list definition that is not a SUBR, even if it does not begin with LAMBDA or NLAMBDA. In other words, EXPRP is not quite as selective as FNTYP.

(CCODEP *FN*) [Function]

Returns T if (FNTYP *FN*) is either CEXPR, CFEXPR, CEXPR*, or CFEXPR*, i.e., second column of FNTYPs; NIL otherwise.

(SUBRP *FN*) [Function]

Returns T if (FNTYP *FN*) is either SUBR, FSUBR, SUBR*, or FSUBR*, i.e., the third column of FNTYPs; NIL otherwise.

(ARGTYPE *FN*) [Function]

FN is the name of a function or its definition. ARGTYPE returns 0, 1, 2, or 3, or NIL if *FN* is not a function. The interpretation of this value is:

0 lambda-spread functions (EXPR, CEXPR, SUBR)

1 nlambda-spread functions (FEXPR, CFEXPR, FSUBR)

2 lambda-nospread functions (EXPR*, CEXPR*, SUBR*)

3 nlambda-nospread functions (FEXPR*, CFEXPR*, FSUBR*)

i.e., ARGTYPE corresponds to the *rows* of FNTYP's.

(NARGS *FN*) [Function]

Returns the number of arguments of *FN*, or NIL if *FN* is not a function. If *FN* is a nospread function, the value of NARGS is 1.

(ARGLIST *FN*) [Function]

Returns the "argument list" for *FN*. Note that the "argument list" is a litatom for nospread functions. Since NIL is a possible value for ARGLIST, an error is generated, ARGS NOT AVAILABLE, if *FN* is not a function.

If *FN* is a compiled function, the argument list is constructed, i.e., each call to ARGLIST requires making a new list. For EXPRs, whose definitions are lists beginning with LAMBDA or NLAMBDA, the argument list is simply CADR of GETD. If *FN* has a list definition, and CAR of the definition is not LAMBDA or NLAMBDA, ARGLIST will check to see if CAR of the definition is a member of LAMBDASPLST (page 15.12). If it is, ARGLIST presumes this is a function object the user is defining via DWIMUSERFORMS (page 15.10), and simply returns CADR of the definition as its argument list. Otherwise ARGLIST generates an error as described above.

(Interlisp-10) If *FN* is a spread SUBR, the ARGLIST returns (U), (U V), (U V W), etc. depending on the number of arguments; if a nospread SUBR, it returns U. This is merely a "feature" of ARGLIST; SUBRs do not actually store the names of their arguments(s) on the stack.

(SMARTARGLIST *FN EXPLAINFLG TAIL*) [Function]

A "smart" version of ARGLIST that tries various strategies to get the arglist of *FN*.

If *FN* is not defined as a function, SMARTARGLIST attempts spelling correction on *FN* by calling FNCHECK (page 15.19), passing *TAIL* to be used for the call to FIXSPELL. If unsuccessful, an error will be generated, *FN* NOT A FUNCTION.

If *FN* is known to the file package (page 11.1) but not loaded in, SMARTARGLIST will obtain the arglist information from the file.

In Interlisp-10, if the HELPSYS help system is installed, SMARTARGLIST may use it to look up the arguments to *FN* in the Interlisp manual files. Specifically, HELPSYS will be used if *EXPLAINFLG*=T and *FN* is a nospread function, or if *FN* is a spread SUBR, regardless of the value of *EXPLAINFLG*. For all other cases, and when HELPSYS is undefined or unsuccessful in finding the arguments, SMARTARGLIST simply returns (ARGLIST *FN*).

In order to avoid repeated calls to HELPSYS, and also to provide the user with an override, SMARTARGLIST stores the arguments returned from HELPSYS on the property list of *FN* under the property ARGNAMES and checks for this property before calling HELPSYS. For spread functions, the argument list itself is stored.

For nospread, the form is (NIL *ARGLIST*$_1$. *ARGLIST*$_2$) where *ARGLIST*$_1$ is the value of SMARTARGLIST when *EXPLAINFLG*=T, and *ARGLIST*$_2$ the value when *EXPLAINFLG*=NIL. For example, (GETPROP 'DEFINEQ 'ARGNAMES) = (NIL (X1 XI ... XN) . X).

SMARTARGLIST is used by BREAK (page 10.4) and ADVISE (page 10.9) with *EXPLAINFLG*=NIL for constructing equivalent EXPR definitions, and by the programmer's assistant command ?= (page 9.5), with *EXPLAINFLG*=T.

5.2 FUNCTION DEFINITION

Function definitions are stored in a "function definition cell" associated with each litatom. This cell is directly accessible via the two functions PUTD and GETD, but it is usually easier to define functions with DEFINEQ (page 5.9).

(GETD *FN*) [Function]

Returns the function definition of *FN*. Returns NIL if *FN* is not a litatom, or has no definition.

GETD of a compiled function constructs a pointer to the definition, with the result that two successive calls do not produce EQ results. EQP or EQUAL must be used to compare compiled definitions.

(Interlisp-10) GETD of a SUBR performs a CONS.

(FGETD *FN*) [Function]

Faster version of GETD. Interpreted, generates an error, BAD ARGUMENT - FGETD, if *FN* is not a litatom.

FGETD is intended primarily to check whether a function *has* a definition, rather than to obtain the definition. Therefore, in Interlisp-10, FGETD of a SUBR returns just the address of the function definition, not the dotted pair returned by GETD, thereby saving the CONS.

(PUTD *FN DEF* —) [Function]

Puts *DEF* into *FN*'s function cell, and returns *DEF*. Generates an error, ARG NOT LITATOM, if *FN* is not a litatom. Generates an error, ILLEGAL ARG, if *DEF* is a string, number, or a litatom other than NIL.

(PUTDQ *FN DEF*) [NLambda Function]

Nlambda version of PUTD; both arguments are unevaluated. Returns *FN*.

(PUTDQ? *FN DEF*) [NLambda Function]

If *FN* is not defined, same as PUTDQ. Otherwise, does nothing and returns NIL.

(MOVD *FROM TO COPYFLG*) [Function]

Moves the definition of *FROM* to *TO*, i.e., redefines *TO*. If *COPYFLG*=T, a COPY of the definition of *FROM* is used. *COPYFLG*=T is only meaningful for EXPRs, although MOVD works for compiled functions and SUBRs as well. MOVD returns

TO.

(MOVD? *FROM TO COPYFLG*) [Function]

If *TO* is not defined, same as (MOVD *FROM TO COPYFLG*). Otherwise, does nothing and returns NIL.

(DEFINEQ x_1 x_2 ⋯ x_N) [NLambda NoSpread Function]

DEFINEQ is the function normally used for defining functions. It takes an indefinite number of arguments which are not evaluated. Each x_i must be a list defining one function, of the form (*NAME DEFINITION*). For example:

```
(DEFINEQ (DOUBLE (LAMBDA (X) (IPLUS X X))) )
```

The above expression will define the function DOUBLE with the EXPR definition (LAMBDA (X) (IPLUS X X)). x_i may also have the form (*NAME ARGS . DEF-BODY*), in which case an appropriate Lambda EXPR will be constructed. Therefore, the above expression is exactly the same as:

```
(DEFINEQ (DOUBLE (X) (IPLUS X X)) )
```

Note that this alternate form can only be used for Lambda functions. The first form must be used to define an Nlambda function.

DEFINEQ returns a list of the names of the functions defined.

(DEFINE *X* —) [Function]

Lambda-spread version of DEFINEQ. Each element of the list *X* is itself a list either of the form (*NAME DEFINITION*) or (*NAME ARGS . DEF-BODY*). DEFINE will generate an error, INCORRECT DEFINING FORM, on encountering an atom where a defining list is expected.

Note: DEFINE and DEFINEQ will operate correctly if the function is already defined and BROKEN, ADVISED, or BROKEN-IN.

For expressions involving type-in only, if the time stamp facility is enabled (page 17.60), both DEFINE and DEFINEQ will stamp the definition with the user's initials and date.

DFNFLG [Variable]

DFNFLG is a global variable that effects the operation of DEFINE (and DEFINEQ, which calls DEFINE). If DFNFLG=NIL, an attempt to *redefine* a function *FN* will cause DEFINE to print the message (*FN* REDEFINED) and to save the old definition of *FN* using SAVEDEF before redefining it, except if the old and new definitions are the same (i.e. EQUAL), the effect is simply a no-op. If DFNFLG=T, the function is simply redefined. If DFNFLG=PROP or ALLPROP, the new definition is stored on the property list under the property EXPR. ALLPROP affects the operation of RPAQQ and RPAQ (page 11.37). DFNFLG is initially NIL.

DFNFLG is reset by LOAD (page 11.4) to enable various ways of handling the defining of functions and setting of variables when loading a file. For most applications, the user will not reset DFNFLG directly.

(SAVEDEF *FN*) [Function]

Saves the definition of *FN* on its property list under the property EXPR, CODE, or SUBR depending on its FNTYP. Returns the property name used. If (GETD *FN*) is non-NIL, but (FNTYP *FN*)=NIL, SAVEDEF saves the definition on the property name LIST. This situation can arise when a function is redefined which was originally defined with LAMBDA misspelled or omitted.

If *FN* is a list, SAVEDEF operates on each function in the list, and returns a list of the individual values.

(UNSAVEDEF *FN PROP*) [Function]

Restores the definition of *FN* from its property list under property *PROP* (see SAVEDEF above). Returns *PROP*. If nothing is saved under *PROP*, and *FN* is defined, returns (*PROP* NOT FOUND), otherwise generates an error, NOT A FUNCTION.

If *PROP* is not given, i.e., NIL, UNSAVEDEF looks under the properties EXPR, CODE, and SUBR, in that order. The value of UNSAVEDEF is the property-name, or if nothing is found and *FN* is a function, the value is (NOTHING FOUND); otherwise generates an error, NOT A FUNCTION.

If DFNFLG=NIL, the current definition of *FN*, if any, is saved using SAVEDEF. Thus one can use UNSAVEDEF to switch back and forth between two definitions of the same function, keeping one on its property list and the other in the function definition cell.

If *FN* is a list, UNSAVEDEF operates on each function of the list, and its value is a list of the individual values.

Both SAVEDEF and UNSAVEDEF are redefined in more general terms (see page 11.18) to operate on typed definitions of which a function definition is but one example. Thus, their actual argument lists in Interlisp are different than given here. However, when their extra arguments are defaulted to NIL, they operate as described above.

5.3 FUNCTION EVALUATION

Usually, function application is done automatically by the Interlisp interpreter. If a form is typed into Interlisp whose CAR is a function, this function is applied to the arguments in the CDR of the form. These arguments are evaluated or not, and bound to the function parameters, as determined by the type of the function, and the body of the function is evaluated. This sequence is repeated as each form in the body of the function is evaluated.

There are some situations where it is necessary to explicitly call the evaluator, and Interlisp supplies a number of functions that will do this. These functions take "functional arguments", which may either be litatoms with function definitions, or EXPR forms such as (LAMBDA (X) ···), or FUNARG expressions (see page 5.15).

The following functions are useful when one wants to supply a functional argument which will always return NIL, T, or 0.

(NILL) [NoSpread Function]

Returns NIL.

(TRUE) [NoSpread Function]

Returns T.

(ZERO) [NoSpread Function]

Returns 0.

Note: When using EXPR expressions as functional arguments, they should be enclosed within the function FUNCTION (page 5.15) rather than QUOTE, so that they will be compiled as separate functions. FUNCTION can also be used to create FUNARG expressions, which can be used to solve some problems with referencing free variables, or to create functional arguments which carry "state" along with them.

(EVAL *x* —) [Function]

EVAL evaluates the expression *x* and returns this value, i.e., EVAL provides a way of calling the Interlisp interpreter. Note that EVAL is itself a lambda function, so *its* argument is first evaluated, e.g.,

```
←(SETQ FOO '(ADD1 3))
(ADD1 3)
←(EVAL FOO)
4
←(EVAL 'FOO)
(ADD1 3)
```

Interlisp functions can either evaluate or not evaluate these arguments. For those cases where it is desirable to specify arguments unevaluated, one may use the QUOTE function:

(QUOTE *x*) [NLambda NoSpread Function]

This is a function that prevents its arguments from being evaluated. Its value is *x* itself, e.g., (QUOTE FOO) is FOO.

Note: Since giving QUOTE more than one argument is almost always a parentheses error, and one that would otherwise go undetected, QUOTE itself generates an error in this case, PARENTHESIS ERROR.

(KWOTE *x*) [Function]

Value is an expression which when evaluated yields *x*. If *x* is NIL or a number, this is *x* itself. Otherwise, (LIST (QUOTE QUOTE) *x*). For example, if the value of X is A and the value of Y is B, then (KWOTE (CONS X Y)) = (QUOTE (A . B)).

(DEFEVAL *TYPE FN*) [Function]

Specifies how a datum of a particular type is to be evaluated.[1] Intended primarily for user defined data types, but works for all data types except lists, literal atoms, and numbers. *TYPE* is a type name. *FN* is a function object, i.e. name of a function or a lambda expression. Whenever the interpreter encounters a datum of the indicated type, *FN* is applied to the datum and its value returned as the result of the evaluation. DEFEVAL returns the previous evaling function for this type. If *FN*=NIL, DEFEVAL returns the current evaling function without changing it. If

[1] COMPILETYPELST (page 12.9) permits the user to specify how a datum of a particular type is to be compiled.

FN=T, the evaling function is set back to the system default (which for all data types except lists is to return the datum itself).

(APPLY *FN ARGLIST* —) [Function]

Applies the function *FN* to the arguments in the list *ARGLIST*, and returns its value. APPLY is a lambda function, so its arguments are evaluated, but the individual elements of *ARGLIST* are not evaluated. Therefore, lambda and nlambda functions are treated the same by APPLY; lambda functions take their arguments from *ARGLIST* without evaluating them. Note that *FN* may still explicitly evaluate one or more of its arguments itself, as SETQ does. Thus, (APPLY 'SETQ '(FOO (ADD1 3))) will set FOO to 4, whereas (APPLY 'SET '(FOO (ADD1 3))) will set FOO to the expression (ADD1 3).

APPLY can be used for manipulating EXPRs, for example:

```
←(APPLY '(LAMBDA (X Y) (ITIMES X Y))
        '(3 4))
12
```

(APPLY* *FN ARG*$_1$ *ARG*$_2$ ⋯ *ARG*$_N$) [NoSpread Function]

Nospread version of APPLY, equivalent to (APPLY *FN* (LIST *ARG*$_1$ *ARG*$_2$ ⋯ *ARG*$_N$)).

(EVALA *X A*) [Function]

Simulates a-list evaluation as in LISP 1.5. *X* is a form, *A* is a list of the form:

((*NAME*$_1$. *VAL*$_1$) (*NAME*$_2$. *VAL*$_2$) ⋯ (*NAME*$_N$. *VAL*$_N$))

The variable names and values in *A* are "spread" on the stack, and then *X* is evaluated. Therefore, any variables appearing free in *X*, that also appears as CAR of an element of *A* will be given the value in the CDR of that element.

The functions below are used to evaluate a form or apply a function repeatedly. RPT, RPTQ, and FRPTQ evaluate a given form a specified number of times. MAP, MAPCAR, MAPLIST, etc. apply a given function repeatedly to different elements of a list, possibly constructing another list. These functions allow efficient iterative computations, but they are difficult to use. For programming iterative computations, it is usually better to use the CLISP Iterative Statement facility (page 4.5), which provides a more general and complete facility for expressing iterative statements. Whenever possible, CLISP translates iterative statements into expressions using the functions below, so there is no efficiency loss.

(RPT *N FORM*) [Function]

Evaluates the expression *FORM*, *N* times. Returns the value of the last evaluation. If $N \leq 0$, *FORM* is not evaluated, and RPT returns NIL.

Before each evaluation, the local variable RPTN is bound to the number of evaluations yet to take place. This variable can be referenced within *FORM*. For example, (RPT 10 '(PRINT RPTN)) will print the numbers 10, 9, ⋯ 1, and return 1.

(RPTQ *N FORM*$_1$ *FORM*$_2$ ⋯ *FORM*$_N$) [NLambda NoSpread Function]

Nlambda-nospread version of RPT: *N* is evaluated, *FORM*$_i$ are not. Returns the value of the last evaluation of *FORM*$_N$.

(FRPTQ *N* *FORM*$_1$ *FORM*$_2$ ··· *FORM*$_N$) [NLambda NoSpread Function]

Faster version of RPTQ. Does not bind RPTN.

(MAP *MAPX* *MAPFN1* *MAPFN2*) [Function]

If *MAPFN2* is NIL, MAP applies the function *MAPFN1* to successive tails of the list *MAPX*. That is, first it computes (*MAPFN1* *MAPX*), and then (*MAPFN1* (CDR *MAPX*)), etc., until *MAPX* becomes a non-list. If *MAPFN2* is provided, (*MAPFN2* *MAPX*) is used instead of (CDR *MAPX*) for the next call for *MAPFN1*, e.g., if *MAPFN2* were CDDR, alternate elements of the list would be skipped. MAP returns NIL.

(MAPC *MAPX* *MAPFN1* *MAPFN2*) [Function]

Identical to MAP, except that (*MAPFN1* (CAR *MAPX*)) is computed at each iteration instead of (*MAPFN1* *MAPX*), i.e., MAPC works on elements, MAP on tails. MAPC returns NIL.

(MAPLIST *MAPX* *MAPFN1* *MAPFN2*) [Function]

Successively computes the same values that MAP would compute, and returns a list consisting of those values.

(MAPCAR *MAPX* *MAPFN1* *MAPFN2*) [Function]

Computes the same values that MAPC would compute, and returns a list consisting of those values, e.g., (MAPCAR X 'FNTYP) is a list of FNTYPs for each element on X.

(MAPCON *MAPX* *MAPFN1* *MAPFN2*) [Function]

Computes the same values as MAP and MAPLIST but NCONCs these values to form a list which it returns.

(MAPCONC *MAPX* *MAPFN1* *MAPFN2*) [Function]

Computes the same values as MAPC and MAPCAR, but NCONCs the values to form a list which it returns.

Note that MAPCAR creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. MAPCONC is used when there are a *variable* number of elements (including none) to be inserted at each iteration. Examples:

```
(MAPCONC '(A B C NIL D NIL)
         '(LAMBDA (Y) (if (NULL Y) then NIL else (LIST Y))))
  ==>  (A B C D)
```

This MAPCONC returns a list consisting of *MAPX* with all NILs removed.

```
(MAPCONC '((A B) C (D E F) (G) H I)
         '(LAMBDA (Y) (if (LISTP Y) then Y else NIL)))
  ==>  (A B D E F G)
```

This MAPCONC returns a linear list consisting of all the lists on *MAPX*.

Since MAPCONC uses NCONC to string the corresponding lists together, in this example the original list will be altered to be ((A B D E F G) C (D E F G) (G) H I). If this is an undesirable side effect, the functional argument to MAPCONC should return instead a top level copy of the lists, i.e. (LAMBDA (Y) (if (LISTP Y) then (APPEND Y) else NIL))).

(MAP2C *MAPX MAPY MAPFN1 MAPFN2*) [Function]

Identical to MAPC except *MAPFN1* is a function of two arguments, and (*MAPFN1* (CAR *MAPX*) (CAR *MAPY*)) is computed at each iteration. Terminates when either *MAPX* or *MAPY* is a non-list.

MAPFN2 is still a function of one argument, and is applied twice on each iteration; (*MAPFN2 MAPX*) gives the new *MAPX*, (*MAPFN2 MAPY*) the new *MAPY*. CDR is used if *MAPFN2* is not supplied, i.e., is NIL.

(MAP2CAR *MAPX MAPY MAPFN1 MAPFN2*) [Function]

Identical to MAPCAR except *MAPFN1* is a function of two arguments and (*MAPFN1* (CAR *MAPX*) (CAR *MAPY*)) is used to assemble the new list. Terminates when either *MAPX* or *MAPY* is a non-list.

(SUBSET *MAPX MAPFN1 MAPFN2*) [Function]

Applies *MAPFN1* to elements of *MAPX* and returns a list of those elements for which this application is non-NIL, e.g.,

(SUBSET '(A B 3 C 4) 'NUMBERP) = (3 4).

MAPFN2 plays the same role as with MAP, MAPC, et al.

(EVERY *EVERYX EVERYFN1 EVERYFN2*) [Function]

Returns T if the result of applying *EVERYFN1* to each element in *EVERYX* is true, otherwise NIL. For example, (EVERY '(X Y Z) 'ATOM) => T.

EVERY operates by evaluating (*EVERYFN1* (CAR *EVERYX*) *EVERYX*). The second argument is passed to *EVERYFN1* so that it can look at the next element on *EVERYX* if necessary. If *EVERYFN1* yields NIL, EVERY immediately returns NIL. Otherwise, EVERY computes (*EVERYFN2 EVERYX*), or (CDR *EVERYX*) if *EVERYFN2*=NIL, and uses this as the "new" *EVERYX*, and the process continues. For example, (EVERY *X* 'ATOM 'CDDR) is true if every *other* element of *X* is atomic.

(SOME *SOMEX SOMEFN1 SOMEFN2*) [Function]

Returns the tail of *SOMEX* beginning with the first element that satisfies *SOMEFN1*, i.e., for which *SOMEFN1* applied to that element is true. Value is NIL if no such element exists. (SOME X '(LAMBDA (Z) (EQUAL Z Y))) is equivalent to (MEMBER Y X). SOME operates analogously to EVERY. At each stage, (*SOMEFN1* (CAR *SOMEX*) *SOMEX*) is computed, and if this is not NIL, *SOMEX* is returned as the value of SOME. Otherwise, (*SOMEFN2 SOMEX*) is computed, or (CDR *SOMEX*) if *SOMEFN2*=NIL, and used for the next *SOMEX*.

(NOTANY *SOMEX SOMEFN1 SOMEFN2*) [Function]

(NOT (SOME *SOMEX SOMEFN1 SOMEFN2*))

(NOTEVERY *EVERYX EVERYFN1 EVERYFN2*) [Function]

(NOT (EVERY *EVERYX EVERYFN1 EVERYFN2*))

(MAPRINT *LST FILE LEFT RIGHT SEP PFN LISPXPRINTFLG*) [Function]

A general printing function. It cycles through *LST* applying *PFN* (or PRIN1 if *PFN* not given) to each element of *LST*. Between each application, MAPRINT performs

PRIN1 of *SEP* (or " " if *SEP*=NIL). If *LEFT* is given, it is printed (using PRIN1) initially; if *RIGHT* is given it is printed (using PRIN1) at the end.

For example, (MAPRINT X NIL '%('%)) is equivalent to PRIN1 for lists. To print a list with commas between each element and a final "." one could use (MAPRINT X T NIL '%. '%,).

If *LISPXPRINTFLG*=T, LISPXPRIN1 (page 8.20) is used instead of PRIN1.

5.4 FUNCTIONAL ARGUMENTS

When using functional arguments, the following function is very useful:

(FUNCTION *FN ENV*) [NLambda Function]

If *ENV*=NIL, FUNCTION is the same as QUOTE, except that it is treated differently when compiled. Consider the function definition:

```
(DEFINEQ (FOO ···
              (FIE LST (FUNCTION (LAMBDA (Z) (ITIMES Z Z))))
) )
```

FOO calls the function FIE with the value of LST and the EXPR expression (LAMBDA (Z) (LIST (CAR Z))).

If FOO is run interpreted, it doesn't make any difference whether FUNCTION or QUOTE is used. However, when FOO is compiled, if FUNCTION is used the compiler will define and compile the EXPR as an auxiliary function (See page 12.8). The compiled EXPR will run considerably faster, which can make a big difference if it is applied repeatedly.

Note: Compiling FUNCTION will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (MAPCAR, MAPLIST, etc.).

If *ENV* is not NIL, it can be a list of variables that are (presumably) used freely by *FN*. In this case, the value of FUNCTION is an expression of the form (FUNARG *FN POS*), where *POS* is a stack pointer to a frame that contains the variable bindings for those variables on *ENV*. *ENV* can also be a stack pointer itself, in which case the value of FUNCTION is (FUNARG *FN ENV*). Finally, *ENV* can be an atom, in which case it is evaluated, and the value interpreted as described above.

As explained above, one of the possible values that FUNCTION can return is the form (FUNARG *FN POS*), where *FN* is a function and *POS* is a stack pointer. FUNARG is not a function itself. Like LAMBDA and NLAMBDA, it has meaning and is specially recognized by Interlisp only in the context of applying a function to arguments. In other words, the expression (FUNARG *FN POS*) is used exactly like a function. When a FUNARG expression is applied or is CAR of a form being EVAL'ed, the APPLY or EVAL takes place in the access environment specified by *ENV* (see page 7.1). Consider the following example:

```
← (DEFINEQ (DO.TWICE (FN VAL)
               (APPLY* FN (APPLY* FN VAL))) )
(DO.TWICE)
← (DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X X))]
            5)
20
← (SETQ VAL 1)
1
← (DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X VAL))]
            5)
20
← (DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X VAL)) (VAL)]
            5)
7
```

DO.TWICE is defined to apply a function FN to a value VAL, and apply FN again to the value returned; in other words it calculates (FN (FN VAL)). Given the EXPR expression (LAMBDA (X) (IPLUS X X)), which doubles a given value, it correctly calculates (FN (FN 5)) = (FN 10) = 20. However, when given (LAMBDA (X) (IPLUS X VAL)), which should add the value of the global variable VAL to the argument X, it does something unexpected, returning 20 again, rather than 5+1+1 = 7. The problem is that when the EXPR is evaluated, it is evaluated in the context of DO.TWICE, where VAL is bound to the second argument of DO.TWICE, namely 5. In this case, one solution is to use the *ENV* argument to FUNCTION to construct a FUNARG expression which contains the value of VAL at the time that the FUNCTION is executed. Now, when (LAMBDA (X) (IPLUS X VAL)) is evaluated, it is evaluated in an environment where the global value of VAL is accessable. Admittedly, this is a somewhat contrived example (it would be easy enough to change the argument names to DO.TWICE so there would be no conflict), but this situation arises occasionally with large systems of programs that construct functions, and pass them around.

Note: System functions with functional arguments (APPLY, MAPCAR, etc.) are compiled so that their arguments are local, and not accessable (see page 12.4). This reduces problems with conflicts with free variables used in functional arguments.

FUNARG expressions can be used for more than just circumventing the clashing of variables. For example, a FUNARG expression can be returned as the value of a computation, and then used "higher up". Furthermore, if the function in a FUNARG expression *sets* any of the variables contained in the frame, only the frame would be changed. For example, consider the following function:

```
(MAKECOUNTER (CNT)
   (FUNCTION [LAMBDA NIL
                (PROG1 CNT (SETQ CNT (ADD1 CNT]
             (CNT)))
```

The function MAKECOUNTER returns a FUNARG that increments and returns the previous value of the counter CNT. However, this is done within the environment of the call to MAKECOUNTER where FUNCTION was executed, which the FUNARG expression "carries around" with it, even after MAKECOUNTER has finished executing. Note that each call to MAKECOUNTER creates a FUNARG expression with a new, independent environment, so that multiple counters can be generated and used:

```
← (SETQ C1 (MAKECOUNTER 1))
(FUNARG (LAMBDA NIL (PROG1 CNT (SETQ CNT (ADD1 CNT)))) #1,13724/*FUNARG)
← (APPLY C1)
1
```

```
← (APPLY C1)
2
← (SETQ C2 (MAKECOUNTER 17))
(FUNARG (LAMBDA NIL (PROG1 CNT (SETQ CNT (ADD1 CNT)))) #1,13736/*FUNARG)
← (APPLY C2)
17
← (APPLY C2)
18
← (APPLY C1)
3
← (APPLY C2)
19
```

By creating a FUNARG expression with FUNCTION, a program can create a function object which has updateable binding(s) associated with the object which last *between* calls to it, but are only accessible through that instance of the function. For example, using the FUNARG device, a program could maintain two different instances of the same random number generator in different states, and run them independently.

Note: In Interlisp-10, environment switching is expensive because it is a shallow-binding system (see page 7.1), so this may restrict the applications of FUNARG expressions.

5.5 MACROS

Macros provide an alternative way of specifying the action of a function. Whereas function definitions are evaluated with a "function call", which involves binding variables and other housekeeping tasks, macros are evaluated by *translating* one Interlisp form into another, which is then evaluated.

A litatom may have both a function definition and a macro definition. When a form is evaluated by the interpreter, if the CAR has a function definition, it is used (with a function call), otherwise if it has a macro definition, then that is used. However, when a form is compiled, the CAR is checked for a macro definition first, and only if there isn't one is the function definition compiled. This allows functions that behave differently when compiled and interpreted. For example, it is possible to define a function that, when interpreted, has a function definition that is slow and has a lot of error checks, for use when debugging a system. This function could also have a macro definition that defines a fast version of the function, which is used when the debugged system is compiled.

Macro definitions are represented by lists that are stored on the property list of a litatom. Macros are often used for functions that should be compiled differently in different Interlisp implementations, and the exact property name a macro definition is stored under determines whether it should be used in a particular implementation. The global variable MACROPROPS contains a list of all possible macro property names which should be saved by the MACROS file package command. Typical macro property names are 10MACRO for Interlisp-10, DMACRO for Interlisp-D,[2] and MACRO for "implementation independent" macros. The global variable COMPILERMACROPROPS is a list of macro property names. Interlisp determines whether a litatom has a macro definition by checking these property names, in order, and

[2] also VAXMACRO for Interlisp-VAX, and JMACRO for Interlisp-Jerico.

using the first non-NIL property value as the macro definition. In Interlisp-D this list contains DMACRO and MACRO in that order so that DMACROs will override the implementation-independent MACRO properties. In general, use a DMACRO property for macros that are to be used only in Interlisp-D, use 10MACRO for macros that are to be used only in Interlisp-10, and use MACRO for macros that are to affect both systems.

Macro definitions can take the following forms:

(LAMBDA ···) or (NLAMBDA ···)

A function can be made to compile open by giving it a macro definition of the form (LAMBDA ···) or (NLAMBDA ···), e.g., (LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) for ABS. The effect is as if the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as a lambda or nlambda expression. This saves the time necessary to call the function at the price of more compiled code generated in-line.

(NIL *EXPRESSION*) or (*LIST* *EXPRESSION*)

"Substitution" macro. Each argument in the form being evaluated or compiled is substituted for the corresponding atom in *LIST*, and the result of the substitution is used instead of the form. For example, if the macro definition of ADD1 is ((X) (IPLUS X 1)), then, (ADD1 (CAR Y)) is compiled as (IPLUS (CAR Y) 1).

Note that ABS could be defined by the substitution macro ((X) (COND ((GREATERP X 0) X) (T (MINUS X)))). In this case, however, (ABS (FOO X)) would compile as

```
(COND ((GREATERP (FOO X) 0)
        (FOO X))
       (T (MINUS (FOO X))))
```

and (FOO X) would be evaluated two times. (Code to evaluate (FOO X) would be generated three times.)

(OPENLAMBDA *ARGS* *BODY*)

This is a cross between substitution and LAMBDA macros. When the compiler processes an OPENLAMBDA, it attempts to substitute the actual arguments for the formals wherever this preserves the frequency and order of evaluation that would have resulted from a LAMBDA expression, and produces a LAMBDA binding only for those that require it.

T

When a macro definition is the atom T, it means that the compiler should ignore the macro, and compile the function definition; this is a simple way of turning off other macros. For example, the user may have a function that runs in both Interlisp-D and Interlisp-10, but has a macro definition that should only be used when compiling in Interlisp-10. If the MACRO property has the macro specification, a DMACRO of T will cause it to be ignored by the Interlisp-D compiler. Note that this DMACRO would not be necessary if the macro were specified by a 10MACRO instead of a MACRO.

(= . *OTHER-FUNCTION*)

A simple way to tell the compiler to compile one function exactly as it would compile another. For example, when compiling in Interlisp-D, FRPLACAs are treated as RPLACAs. This is achieved by having FRPLACA have a DMACRO of (= . RPLACA).

(*LITATOM* *EXPRESSION*)

If a macro definition begins with a litatom other than those given above, this allows *computation* of the Interlisp expression to be evaluated or compiled in place of the form. *LITATOM* is bound to the CDR of the calling form, *EXPRESSION* is evaluated, and the result of this evaluation is evaluated or compiled in place of the form. For example, LIST could be compiled using the computed macro:

```
[X (LIST 'CONS
         (CAR X)
         (AND (CDR X)
              (CONS 'LIST
                    (CDR X]
```

This would cause (LIST X Y Z) to compile as (CONS X (CONS Y (CONS Z NIL))). Note the recursion in the macro expansion.

If the result of the evaluation is the litatom IGNOREMACRO, the macro is ignored and the compilation of the expression proceeds as if there were no macro definition. If the litatom in question is normally treated specially by the compiler (CAR, CDR, COND, AND, etc.), and also has a macro, if the macro expansion returns IGNOREMACRO, the litatom will still be treated specially.

In Interlisp-10, if the result of the evaluation is the atom INSTRUCTIONS, no code will be generated by the compiler. It is then assumed the evaluation was done for effect and the necessary code, if any, has been added. This is a way of giving direct instructions to the compiler if you understand it.

Note: It is often useful, when constructing complex macro expressions, to use the BQUOTE facility (see page 6.39).

The following function is quite useful for debugging macro definitions:

(EXPANDMACRO *FORM QUIETFLG* —) [Function]

Takes a form whose CAR has a macro definition and expands the form as it would be compiled. The result is prettyprinted, unless *QUIETFLG*=T, in which case the result is simply returned.

5.5.1 MACROTRAN

Interpreted macros are implemented by the function MACROTRAN. When the interpreter encounters a form CAR of which is an undefined function,[3] MACROTRAN is called. If CAR of the form has a macro definition, the macro is expanded, and the result of this expansion is evaluated in place of the original form. CLISPTRAN (page 16.19) is used to save the result of this expansion so that the expansion only has to be done once. On subsequent occasions, the translation (expansion) is retrieved from CLISPARRAY the same as for other CLISP constructs; MACROTRAN never even has to be invoked.

Sometimes, macros contain calls to functions that assume that the macro is being compiled. The variable SHOULDCOMPILEMACROATOMS is a list of functions that should be compiled to work correctly (initially (OPCODES) in Interlisp-D, (ASSEMBLE LOC) in Interlisp-10). UNSAFEMACROATOMS is a list

[3] In other words, if you have a macro on FOO, then typing (FOO 'A 'B) will work, but FOO(A B) will not work.

of functions which effect the operation of the compiler, so such macro forms shouldn't even be expanded except by the compiler (initially NIL in Interlisp-D, (C2EXP STORIN CEXP COMP) in Interlisp-10). If MACROTRAN encounters a macro containing calls to functions on these two lists, instead of the macro being expanded, a dummy function is created with the form as its definition, and the dummy function is then compiled. A form consisting of a call to this dummy function with no arguments is then evaluated in place of the original form, and CLISPTRAN is used to save the translation as described above. There are some situations for which this procedure is not amenable, e.g. a GO inside the form which is being compiled will cause the compiler to give an UNDEFINED TAG error message because it is not compiling the entire function, just a part of it.

Note: MACROTRAN is an entry on DWIMUSERFORMS (page 15.10) and thus will not work if DWIM is not enabled.

CHAPTER 6

INPUT/OUTPUT

6.1 FILES

All input/output functions in Interlisp can specify their source/destination file with an optional extra argument, which is the name of the file, given as a litatom. These functions generally require that the file be *open*. Files are opened and manipulated by the functions described below. The name T designates terminal input and output, and is always considered open. It is also possible to supply a string as an input "file", without needing to open it; input operations remove successive characters from the string. Note that because of this feature, file names must always be specified as litatoms, not strings.

(OPENFILE *FILE ACCESS RECOG BYTESIZE MACHINE.DEPENDENT.PARAMETERS*) [Function]

Opens *FILE* with access rights as specified by *ACCESS*, one of INPUT, OUTPUT, BOTH, or APPEND, and returns the full name of the file. Causes error FILE NOT FOUND if *FILE* is not recognized by the file system, or other errors if *FILE* is recognized but cannot be opened, e.g. FILE WON'T OPEN if the file is already opened by someone else or is protected against the operation, FILE SYSTEM RESOURCES EXCEEDED if there is no more room in the file system.

For *ACCESS*=INPUT, only input operations are permitted on the file; for *ACCESS*=OUTPUT or *ACCESS*=APPEND, only output operations are permitted. Note: in Interlisp-10 and Interlisp-D, *ACCESS*=OUTPUT implies that one intends to write a new or different file, even if a version number was specified and the corresponding file already exists. Thus any previous contents of the file are discarded, and the file is empty immediately after the OPENFILE. If it is desired to write on an already existing file while preserving the old contents, the file must be opened for access BOTH or APPEND.

RECOG specifies the recognition mode of *FILE*, as described on page 6.4. If *RECOG*=NIL, it defaults according to the value of *ACCESS*: for *ACCESS*=INPUT, *RECOG*=OLD is used; for *ACCESS*=OUTPUT, *RECOG*=NEW is used; for the other values of *ACCESS*, *RECOG*=OLD/NEW is used.

BYTESIZE, if supplied, is the byte size in which to open the file. If *BYTESIZE*=NIL, the bytesize used is the default for the implementation (8 for Interlisp-D, 7 for Interlisp-10).

MACHINE.DEPENDENT.PARAMETERS is a list specifying additional opening parameters. In Interlisp-10, this list may contain the following litatoms:

WAIT Wait if file is busy.

DON'T.CHANGE.DATE

Don't change the access dates.

THAWED Open file in "thawed" mode.

In Interlisp-D, *MACHINE.DEPENDENT.PARAMETERS* should be a list of pairs (*ATTRIB VALUE*), where *ATTRIB* is any file attribute that the file system is willing to allow the user to set (see SETFILEINFO, page 6.7).

If the *FILE* argument to an input (output) function is not given (has value NIL), the file specified as "primary" for input (output) is used. Normally these are both T, for terminal input and output. However, the primary input or output file may be changed with the functions below.

(INPUT *FILE*) [Function]

Sets *FILE* as the primary input file; returns the name of the old primary input file. *FILE* must be open for input. INPUT can also be given a string as argument, interpreted as described above.

(INPUT) returns the current primary input file, which is not changed.

(OUTPUT *FILE*) [Function]

Sets *FILE* as the primary output file; returns the name of the old primary output file. *FILE* must be open for output. A string cannot be used as an output file.

(OUTPUT) returns the current primary output file, which is not changed.

(INFILE *FILE*) [Function]

Opens *FILE* for input, and sets it as the primary input file. Equivalent to (INPUT (OPENFILE *FILE* 'INPUT 'OLD))

(OUTFILE *FILE*) [Function]

Opens *FILE* for output, and sets it as the primary output file. Equivalent to (OUTPUT (OPENFILE *FILE* 'OUTPUT 'NEW)).

(IOFILE *FILE*) [Function]

(OPENFILE *FILE* 'BOTH 'OLD); opens *FILE* for both input and output. Does not affect the primary input or output file.

(OPENP *FILE ACCESS*) [Function]

If *ACCESS*=NIL, returns the full name of *FILE* if *FILE* is open either for input or for output; otherwise NIL.

If *ACCESS* is INPUT, OUTPUT or BOTH, returns the full name of *FILE* if it is open in that access mode; otherwise NIL.

Note: If *FILE* is not recognized, OPENP returns NIL without generating an error.

(OPENP) returns a list of all files open for input or output, excluding T and the current typescript (dribble) file, if any (page 6.12).

(CLOSEF *FILE*) [Function]

Closes *FILE*. Generates an error, FILE NOT OPEN, if *FILE* is not open. If *FILE* is NIL, it attempts to close the primary input file if other than terminal. Failing that, it attempts to close the primary output file if other than terminal. Failing both, it returns NIL. If it closes any file, it returns the name of that file. If it closes either of the primary files, it resets that primary file to terminal.

WHENCLOSE (page 6.11) allows the user to "advise" CLOSEF to perform various operations when a file is closed.

(CLOSEF? *FILE*) [Function]

Closes *FILE* if it is open, otherwise does nothing. Returns *FILE*.

(CLOSEALL *ALLFLG*) [Function]

Closes all open files, except T and the current typescript file, if any. Returns a list of the files closed.

WHENCLOSE (page 6.11) allows certain files to be "protected" from CLOSEALL. (CLOSEALL T) overrides this protection.

(DELFILE *FILE*) [Function]

Deletes *FILE* if possible. Returns *FILE* if deleted, else NIL.

(RENAMEFILE *OLDFILE NEWFILE*) [Function]

Renames *OLDFILE* to be *NEWFILE*. Returns *NEWFILE* if successful, else NIL.

6.1.1 File Naming and Recognition

In Interlisp, a file name is a literal atom composed of one or more *fields*, separated by suitable punctuation. The precise fields and their interpretation is dependent on the implementation; the functions PACKFILENAME and UNPACKFILENAME (page 6.6) are used to construct and take apart filenames in an implementation-independent way.

Depending on the file system implementation, file names given to input/output functions may be incompletely specified, with the file system handling the task of obtaining a specific file from a partial name, or *recognizing* the file. For example, in file systems that support version numbers, one can call OPENFILE giving a file name without a version number, and the file system will supply a default version number based on the context (opening a new file for output vs. an old file for input). Internally, however, each open file has associated with it a completely-specified filename, one that uniquely identifies the file to the file system in any context. It is this "full" file name that is returned from OPENFILE and other functions that return names of open files. For example, (OPENFILE 'FOO 'OUTPUT) might return <LISP>FOO.;3. Any time that an input/output function is called with a file name other than the full file name, Interlisp must perform recognition on the partial file name in order to determine which open file is intended. Thus if repeated operations are to be performed, it is considerably more efficient to use the full file name returned from OPENFILE than to repeatedly use the possibly incomplete name that was used to open the file.

In Interlisp-10, filenames follow the conventions of the operating system (either TENEX or TOPS-20), i.e., *FILE* can be prefixed by a directory name enclosed in angle brackets, can contain <esc>s or control-F's, and can include suffixes and/or version numbers. When a file is opened for input and no version number is given, the highest existing version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name. The full filename in Interlisp-10 consists of directory, name, extension, and version. In Interlisp-D, it also includes a device or host name in brackets, i.e, {PHYLUM}<LISP>FOO.;3).

The following functions can be used to perform file recognition without opening a file:

Warning: In some implementations of Interlisp (such as Interlisp-D), it may not be possible to determine the full name of a new file without trying to open it. In this case, OUTFILEP *and* FULLNAME *may not always return the correct value. These functions should not be used in general, because the idea "what a file would be named if it were opened" is not well defined in some file systems.*

(INFILEP *FILE*) [Function]

Returns full file name of *FILE* if *FILE* is recognized as specifying the name of an existing file that could potentially be opened for input, NIL otherwise. Recognition is in input context, i.e., in Interlisp-10, if no version number is given, the highest existing version number is returned.

(OUTFILEP *FILE*) [Function]

Similar to INFILEP, except recognition is in output context, i.e., in Interlisp-10, if no version number is given, a version number one higher than the highest existing version number is returned. Roughly speaking, OUTFILEP returns the full name of the file that would be created if OUTFILE were called with the same argument.

A more general version of INFILEP and OUTFILEP is provided by the function FULLNAME:

(FULLNAME *X* *RECOG*) [Function]

If *X* is recognized in the recognition mode specified by *RECOG* as an abbreviation for some file, returns the file's full name, otherwise NIL. *RECOG* can be OLD, meaning choose the (newest) existing version of the file; NEW, meaning make the full file name one which does not yet exist (version number one higher than highest existing version); OLDEST, meaning choose the existing file with the lowest version number; or OLD/NEW, meaning to recognize an existing version if possible, otherwise a new version (useful only for writing a file). *RECOG*=NIL defaults to OLD. For all other values of *RECOG*, generates an error ILLEGAL ARG. If *X* is not a literal atom, generates an error, ARG NOT LITATOM.

For example, INFILEP could be defined as (FULLNAME *FILE* 'OLD) and OUTFILEP as (FULLNAME *FILE* 'NEW).

The *RECOG* argument is used only for defaulting unspecified parts of the filename (in Interlisp-10 and Interlisp-D, the version), not to pass judgment on the specified parts. In particular, *RECOG*=NEW does not require that the file be new. For example, (FULLNAME 'FOO.;2 'NEW) may return <MASINTER>FOO.;2 if that file already exists, even though (FULLNAME 'FOO 'NEW) would default the version to a new number, perhaps returning <MASINTER>FOO.;5.

Note that INFILEP, OUTFILEP and FULLNAME do not open any files, or change the primary files; they are pure predicates. In general they are also only hints, as they do not necessarily imply that the caller has access rights to the file. For example, INFILEP might return non-NIL, but OPENFILE might fail for the same file because the file is read-protected against the user, or the file happens to be open for output by another user at the time. Similarly, OUTFILEP could return non-NIL, but OPENFILE could fail with a FILE SYSTEM RESOURCES EXCEEDED error. Note also that in a multi-user file system, intervening file operations by another user could contradict the information returned by recognition. For example, a file that was INFILEP might be deleted, or between an OUTFILEP and the subsequent OPENFILE, another user might create a new version or delete the highest version, causing the names returned by OUTFILEP and OPENFILE to have different version numbers. Thus, in general, the "truth" about a file can only be obtained by actually opening the file; in particular, creators of files should rely on the name returned from OPENFILE, not from OUTFILEP.

If the file system does not successfully recognize an incomplete file name, a FILE NOT FOUND error is generated (except for INFILEP, OUTFILEP, FULLNAME and OPENP, which in this case return NIL). As described on page 9.16, before a FILE NOT FOUND error occurs, it is intercepted via an entry on ERRORTYPELST, which causes SPELLFILE (page 15.20) to be called. SPELLFILE will search alternate directories and possibly attempt spelling correction on the file name. Only if SPELLFILE is unsuccessful will the error actually occur.

Note that recognition is performed on the user's entire directory, not just the open files, which can result in certain anomalies. Thus, even if only one file is open, say FOO.;1, the name F$ (F<esc>) will not be recognized if the user's directory also contains the file FIE.;1. Similarly, it is possible for a file name that was previously recognized to become ambiguous. For example, a program performs (INFILE 'FOO), opening FOO.;1, and reads several expressions from FOO. Then the user interrupts the program, creates a FOO.;2 and reenters his program. Now a call to READ giving it FOO as its *FILE* argument will generate a FILE NOT OPEN error, because FOO will be recognized as FOO.;2.

6.1.2 Manipulating File Names

Different operating systems have different conventions for naming files. However, it is desirable for Interlisp to be as implementation independent as possible. Therefore, all programs that need to reference parts of a filename, or construct new file names from existing ones, should use the functions described below. The implementation of these functions obviously is dependent on the operating system they will run under, but as far as the programs that use them are concerned, they permit expressing operations that are implementation independent.[1]

Every file name is composed of a collection of *fields* which have different semantic interpretations. A *field name* is a literal atom which is the name of a file-name field. Interlisp assumes that NAME and EXTENSION are valid field-names; the implementor is free to allow other fields. In Interlisp-10, allowable field names are: DEVICE, DIRECTORY, NAME, EXTENSION, VERSION, PROTECTION, ACCOUNT, and TEMPORARY. Interlisp-D allows HOST, DIRECTORY, NAME, EXTENSION, and VERSION.

(FILENAMEFIELD *FILENAME FIELDNAME*) [Function]

Returns the contents of the *FIELDNAME* field of *FILENAME*.

(UNPACKFILENAME *FILENAME* —) [Function]

Returns a list of alternating field names and field contents.

Examples from Interlisp-D:

```
← (UNPACKFILENAME 'FOO.BAR)
(NAME FOO EXTENSION BAR)
← (UNPACKFILENAME '{PHYLUM}<SANNELLA>LISP>IMTRAN.DCOM;21)
```

[1] In particular, the Interlisp-10 implementation recognizes file names in both Tenex and TOPS-20 format, and builds new names as appropriate.

```
(HOST PHYLUM DIRECTORY SANNELLA>LISP NAME IMTRAN
EXTENSION DCOM VERSION 21)
```

Examples from Interlisp-10 on Tenex:

```
← (UNPACKFILENAME '<LISP>MAC.COM;3)
(DIRECTORY LISP NAME MAC EXTENSION COM VERSION 3)
← (UNPACKFILENAME 'WORK.;T)
(NAME WORK EXTENSION NIL TEMPORARY T)
```

Note: In Interlisp-10, (UNPACKFILENAME 'DSK:FOO) returns (DEVICE DSK: NAME FOO), i.e. the : is left in. This is so (DEVICE NIL:) may be distinguished from (DEVICE NIL).

(PACKFILENAME *FIELDNAME*$_1$ *FIELDCONTENTS*$_1$ ··· *FIELDNAME*$_N$ *FIELDCONTENTS*$_N$) [NoSpread Function]

Takes a list of alternating field names and field contents (atoms or strings), and returns the corresponding file name. For example, (PACKFILENAME 'DIRECTORY 'LISP 'NAME 'NET) returns <LISP>NET.

If the same field name is given twice, the *first* occurrence is used.

If the "field name" BODY is given, this means that the operand to BODY should itself be unpacked and spliced into the argument list at that point. This is useful for providing default field names, or to change just one field in an existing name.

For example, to take a file name *FILE* and change the DIRECTORY field, perform (PACKFILENAME 'DIRECTORY *NEWDIRECTORY* 'BODY *FILE*). Alternatively, to provide a default for the EXTENSION field, perform (PACKFILENAME 'BODY *FILE* 'EXTENSION *DEFAULT*). This uses *DEFAULT* as the extension unless one is already specified in *FILE*.

Note that a null field is a field that *has* been specified, e.g., if *FILE*=FOO;1 in the above example, the default extension will be used, but if *FILE*=FOO.;1, it will not, because a null extension has been specified.

If the first argument to PACKFILENAME is a list, PACKFILENAME is called on that argument. Thus PACKFILENAME and UNPACKFILENAME operate as inverses.

6.1.3 File Attributes

Any file has a number of "file attributes", such read date, protection, and bytesize. The exact attributes that a file can have is implementation-dependent. The functions GETFILEINFO and SETFILEINFO allow the user to conveniently access file attributes:

(GETFILEINFO *FILE* *ATTRIB*) [Function]

Returns the current setting of the *ATTRIB* attribute of *FILE*. In Interlisp-10, *FILE* may also be a JFN as returned by GTJFN (page 22.22).

In Interlisp-10, GETFILEINFO takes an optional third argument, *SCRATCH*, which is analogous to the third argument of GDATE (page 14.10): a string pointer to reuse for those *ATTRIB*'s which return string values.

(SETFILEINFO *FILE ATTRIB VALUE*) [Function]

Sets the attribute *ATTRIB* of *FILE* to be *VALUE*. SETFILEINFO returns T if it is able to change the attribute *ATTRIB*, and NIL if unsuccessful (some attributes cannot be changed, e.g. it doesn't make sense to change the SIZE of a file without writing something on it).

GETFILEINFO and SETFILEINFO currently recognize the following values for *ATTRIB*:

ACCESS — The current access mode of *FILE* (e.g. INPUT, OUTPUT, BOTH, APPEND) or NIL if *FILE* is not open.

BYTESIZE — The byte size of the file.

LENGTH — The byte position of the end-of-file. Like (GETEOFPTR *FILE*), but *FILE* does not have to be open.

SIZE — The size of *FILE* in pages.

WRITEDATE, READDATE, CREATIONDATE

The date (and time) as a string that *FILE* was respectively last written, last read, and originally created.

IWRITEDATE, IREADDATE, ICREATIONDATE

The respective date in integer form, as IDATE (page 14.10) would return.

TYPE — (Interlisp-D) Either TEXT or BINARY.

OPENBYTESIZE — (Interlisp-10) It is possible that the byte size for the "opening" of a file might differ from the "permanent" bytesize. For example, a 7-bit text file can be opened in 36-bit mode. To obtain the "open" bytesize, use attribute OPENBYTESIZE.

PROTECTION — (Interlisp-10) The "protection code" of *FILE*, as an integer.

DELETED — (Interlisp-10) T if *FILE* is the name of a deleted file, NIL otherwise.

Additional attributes which are available for Interlisp-10 on TOPS-20 systems (DEC release 4 or later) are:

INVISIBLE — T if *FILE* has the invisible attribute, NIL otherwise.

ARCHIVED — T if *FILE* has been archived, NIL otherwise.

OFF-LINE — T if the contents of *FILE* are off-line (i.e. *FILE* has been archived and its contents flushed), NIL otherwise.

(POSITION *FILE N*) [Function]

Returns the column number at which the next character will be read or printed. After a end of line, the column number is 0. If *N* is non-NIL, *resets* the column number to be *N*.

Note that (POSITION *FILE*) is *not* the same as (GETFILEPTR *FILE*) which gives the position in the *file*, not on the *line*.

(LINELENGTH *N FILE*) [Function]

Sets the length of the print line for the output file *FILE* to *N*; returns the former setting of the line length. *FILE* defaults to the primary output file. (LINELENGTH NIL *FILE*) returns the current setting for *FILE*. When a file is first opened, its linelength is set to the value of the variable FILELINELENGTH.

Whenever printing an atom or string would increase a file's position *beyond* the line length of the file, an end of line is automatically inserted first. This action can be defeated by using PRIN3 and PRIN4 (page 6.17).

(SETLINELENGTH *N*) [Function]

If *N* is NIL, interrogates the operating system for the line length of the terminal device, and sets the variable TTYLINELENGTH to this value. If *N* is not NIL, instructs the operating system to set the terminal line length to *N*, and also sets TTYLINELENGTH to *N*. Then, in either case, SETLINELENGTH performs (and returns as its value) (LINELENGTH TTYLINELENGTH T).

Both AFTERSYSOUTFORMS and RESETFORMS (page 8.19) contain a (SETLINELENGTH) so that when the user first runs a SYSOUT, or types control-D, the system obtains the latest information about the terminal.

6.1.4 Randomly Accessible Files

For most applications, files are read starting at their beginning and proceeding sequentially, i.e., the next character read is the one immediately following the last character read. Similarly, files are written sequentially. However, it is also possible to read/write characters at arbitrary positions in a file, essentially treating the file as a large block of auxiliary storage. For example, one application might involve writing an expression at the *beginning* of the file, and then reading an expression from a specified point in its *middle*. This particular example requires the file be open for *both* input and output. However, random file input or output can also be performed on files that have been opened for only input or only output.

Associated with each file is a "file pointer" that points to the location where the next character is to be read from or written to. The file pointer to a file is automatically advanced after each input or output operation. This section describes functions which can be used to *reposition* the file pointer on those files that can be randomly accessed. A file used in this fashion is much like an array in that it has a certain number of addressable locations that characters can be put into or taken from. However, unlike arrays, files can be enlarged. For example, if the file pointer is positioned at the end of a file and anything is written, the file "grows." It is also possible to position the file pointer *beyond* the end of file and then to write. (If the program attempts to *read* beyond the end of file, an END OF FILE error occurs.) In this case, the file is enlarged, and a "hole" is created, which can later be written into. Note that this enlargement only takes place at the *end* of a file; it is not possible to make more room in the middle of a file. In other words, if expression A begins at position 1000, and expression B at 1100, and the program attempts to overwrite A with expression C, which is 200 characters long, part of B will be altered.

The address of a character (byte) is the number of characters (bytes) that precede it in the file, i.e., 0 is the address of the beginning of the file. However, the user should be careful about computing the space needed for an expression, since end-of-line may be represented by a different number of characters in different implementations, even though NCHARS only counts it as one; e.g., end-of-line in Interlisp-10 files is represented as the two characters carriage-return, line-feed. Output functions may also introduce end-of-line's as a result of LINELENGTH considerations.

(GETFILEPTR *FILE*) [Function]

Returns the current position of the file pointer for *FILE*, i.e., the byte address at which the next input/output operation will commence.

(SETFILEPTR *FILE ADR*) [Function]

Sets the file pointer for *FILE* to the position *ADR*; returns *ADR*. The special value *ADR*=-1 is interpreted to mean the address of the end of file.[2]

(GETEOFPTR *FILE*) [Function]

Returns the byte address of the end of file, i.e., the number of bytes in the file. Equivalent to performing (SETFILEPTR *FILE* -1) and returning (GETFILEPTR *FILE*) except that it does not change the current file pointer.

(EOFP *FILE*) [Function]

Returns T if the file pointer to *FILE* is pointing to the end of file; NIL otherwise. *FILE* must be open for (at least) input, or an error is generated, FILE NOT OPEN.

(RANDACCESSP *FILE*) [Function]

Returns *FILE* if *FILE* is randomly accessible, NIL otherwise. The file T is not randomly accessible, nor are the files LPT:, NIL: in Interlisp-10, or certain network file connections in Interlisp-D. *FILE* must be open or an error is generated, FILE NOT OPEN.

(COPYBYTES *SRCFIL DSTFIL START END*) [Function]

Copies bytes (characters) from *SRCFIL* to *DSTFIL*, starting from position *START* and up to but not including position *END*. Both *SRCFIL* and *DSTFIL* must be open. Returns T.

If *END*=NIL, *START* is interpreted as the number of bytes to copy (starting at the current position). If *START* is also NIL, bytes are copied until the end of the file is reached.

(FILEPOS *PATTERN FILE START END SKIP TAIL CASEARRAY*) [Function]

Analogous to STRPOS (page 2.31), but searches a file rather than a string. FILEPOS searches *FILE* for the string *PATTERN*. Search begins at *START* (or the current position of the file pointer, if *START*=NIL), and goes to *END* (or the end of *FILE*, if *END*=NIL). Returns the address of the start of the match, or NIL if not found.

SKIP can be used to specify a character which matches any character in the file. If *TAIL* is T, and the search is successful, the value is the address of the first character *after* the sequence of characters corresponding to *PATTERN*, instead of the starting address of the sequence. In either case, the file is left so that the next i/o operation begins at the address returned as the value of FILEPOS.

[2] Note: If a file is opened for output only, the end of file is initially zero, even if an old file by the same name had existed (see OPENFILE, page 6.1). If a file is opened for both input and output, the initial file pointer is the beginning of the file, but (SETFILEPTR *FILE* -1) will set it to the end of the file. If the file had been opened in append mode by (OPENFILE *FILE* 'APPEND), the file pointer right after opening would be set to the end of the existing file, in which case a SETFILEPTR to position the file at the end would be unnecessary.

CASEARRAY should be a "casearray" that specifies that certain characters should be transformed to other characters before matching. Casearrays are returned by CASEARRAY or SEPRCASE below. *CASEARRAY*=NIL means no transformation will be performed.

A casearray is an implementation-dependent object that is logically an array of character codes with one entry for each possible character. FILEPOS maps each character in the file "through" *CASEARRAY* in the sense that each character code is transformed into the corresponding character code from *CASEARRAY* before matching. Thus if two characters map into the same value, they are treated as equivalent by FILEPOS. CASEARRAY and SETCASEARRAY provide an implementation-independent interface to casearrays.

For example, to search without regard to upper and lower case differences, *CASEARRAY* would be a casearray where all characters map to themselves, except for lower case characters, whose corresponding elements would be the upper case characters. To search for a delimited atom, one could use " *ATOM* " as the pattern, and specify a *CASEARRAY* in which all of the break and separator characters mapped into the same code as space.

For applications calling for extensive file searches, the function FFILEPOS is often faster than FILEPOS.

(FFILEPOS *PATTERN FILE START END SKIP TAIL CASEARRAY*) [Function]

Like FILEPOS, except much faster in most applications.[3] FFILEPOS is an implementation of the Boyer-Moore fast string searching algorithm. This algorithm preprocesses the string being searched for and then scans through the file in steps usually equal to the length of the string. Thus, FFILEPOS speeds up roughly in proportion to the length of the string, e.g., a string of length 10 will be found twice as fast as a string of length 5 in the same position.

Because of certain fixed overheads, it is generally better to use FILEPOS for short searches or short strings.

(CASEARRAY *OLDARRAY*) [Function]

Creates and returns a new casearray, with all elements set to themselves, to indicate the identity mapping.

(Interlisp-D) If *OLDARRAY* is given, it is reused.

(SETCASEARRAY *CASEARRAY FROMCODE TOCODE*) [Function]

Modifies the casearray *CASEARRAY* so that character code *FROMCODE* is mapped to character code *TOCODE*.

(SEPRCASE *CLFLG*) [Function]

Returns a new casearray suitable for use by FILEPOS or FFILEPOS in which all of the break/separators of FILERDTBL are mapped into character code zero. If *CLFLG* is non-NIL, then all CLISP characters will be mapped into this character as well. This is useful for finding a delimited atom in a file. For example, if *PATTERN*

[3] In Interlisp-10, a speedup of 10 to 50 times is typical. In Interlisp-D the speedup is much smaller.

is " FOO ", and (SEPRCASE T) is used for *CASEARRAY*, then FILEPOS will find "(FOO←".

6.1.5 Closing and Reopening Files

The function WHENCLOSE permits the user to associate certain operations with open files that govern how and when the file will be closed, and how the file's status will be restored when a SYSOUT is started up. The user can specify that certain functions will be executed before CLOSEF closes the file and/or after CLOSEF closes the file. The user can make a particular file be invisible to CLOSEALL, so that it will remain open across user invocations of CLOSEALL. Finally, the user can associate a status-saving function with a file which will be called before SYSOUT and which can specify what to do when a SYSOUT is restarted.

(WHENCLOSE *FILE* *PROP*$_1$ *VAL*$_1$ ··· *PROP*$_N$ *VAL*$_N$) [NoSpread Function]

FILE must specify the name of an open file other than T (NIL defaults to the primary input file, if other than T, or primary output file if other than T). The remaining arguments specify properties to be associated with the full name of *FILE*. WHENCLOSE returns the full name of *FILE* as its value.

WHENCLOSE recognizes the following property names:

BEFORE

VAL is a function that CLOSEF will apply to the full name of *FILE* just before it is closed. This might be used, for example, to copy information about the file from an in-core data structure to the file just before it is closed.

AFTER

VAL is a function that CLOSEF will apply to the full name of *FILE* just after it is closed. This capability permits in-core data structures that know about the file to be cleaned up when the file is closed.

BEFORE and AFTER differ in their behavior with respect to SYSOUT. If a file that was open before SYSOUT does not have a STATUS function associated with it that causes the file to be successfully restored after the SYSOUT is started, then the file is considered to have been "closed" by the SYSOUT, and its AFTER function will be executed after the SYSOUT starts.

STATUS

This property provides a way of restoring the status of files when a SYSOUT is resumed. *VAL* is a function that will be applied to the full name of *FILE* just before a SYSOUT. *VAL* is expected to return a list, CAR of which is a function which will be APPLY'd to the CDR when the SYSOUT is started up and which will restore the status of *FILE*. If the value of the APPLY is NIL, it is assumed the file could not be successfully restored, a warning message is printed, and then any AFTER functions associated with the file are executed.

The function PERMSTATUS (page 23.17) produces an expression for re-opening a file after SYSOUT and restoring as many of its attributes as possible.

CLOSEALL

VAL is either YES or NO and determines whether *FILE* will be closed by CLOSEALL (YES) or whether CLOSEALL will ignore it (NO). CLOSEALL uses CLOSEF, so that any AFTER functions will be executed if the file is in fact closed.

EOF

VAL is a function that will be applied to the full name of *FILE* when an end-of-file error occurs, and the ERRORTYPELST entry for that error, if any, returns NIL. The function can examine the context of the error, and can decide whether to close the file, RETFROM some function, or perform some other computation. If the function supplied returns normally (i.e. does not RETFROM some function), the normal error machinery will be invoked (but *FILE* will not be automatically closed if the EOF function did not close it).

Note that multiple AFTER and BEFORE functions may be associated with a file; they are executed in sequence with the most recently associated function executed first. However, a second STATUS specification will supercede an earlier one. The CLOSEALL and EOF values will also override earlier values, so only the last value specified will have an effect. Files are initialized with CLOSEALL - YES, EOF - CLOSEF.

6.1.6 Dribble Files

A dribble file is a "transcript" of all of the input and output on a terminal. The following function enables dribble files for Interlisp:

(DRIBBLE *FILENAME APPENDFLG THAWEDFLG*) [Function]

Opens *FILENAME* and begins recording the typescript. Returns the old dribble file if any, otherwise NIL. If *APPENDFLG*=T, the typescript will be appended to the end of *FILENAME*. If *THAWEDFLG*=T, the file will be opened in "thawed" mode, for those implementations that support it. (DRIBBLE) closes the dribble file. Only one dribble file can be active at any one time, so (DRIBBLE *FILE1*) followed by (DRIBBLE *FILE2*) will cause *FILE1* to be closed.

In Interlisp-D, DRIBBLE opens a dribble file for the current process, recording the input and output for that process. Multiple processes can have separate dribble files open at the same time.

(DRIBBLEFILE) [Function]

Returns the name of the current dribble file, if any, otherwise NIL.

Terminal input is echoed to the dribble file a line buffer at a time. Thus, the typescript produced is somewhat neater than that appearing on the user's terminal, because it does *not* show characters that were erased via control-A or control-Q. Note that the typescript file is *not* included in the list of files returned by (OPENP), nor will it be closed by a call to CLOSEALL or CLOSEF. Only (DRIBBLE) closes the typescript file.

6.2 INPUT FUNCTIONS

Most of the functions described below have an argument *FILE*, which specifies the name of the file on which the operation is to take place. If *FILE* is NIL, the primary input file will be used. If the file argument is a string, input will be taken from that string (and the string pointer reset accordingly).

Most input functions also have a *RDTBL* argument, which specifies the readtable to be used for input. If *RDTBL* is NIL, the primary readtable will be used. Readtables are described on page 6.32.

Note: in all Interlisp-10 symbolic files, end-of-line is indicated by the characters carriage-return and line-feed in that order. Accordingly, on input from files, Interlisp-10 skips all line-feeds that immediately follow carriage-returns. On input from the terminal, Interlisp echos a line-feed whenever a carriage-return is input.

When reading from the terminal, the input is buffered a line at a time (unless buffering has been inhibited by `(CONTROL T)`, or the input is being read by `READC` or `PEEKC`) and can be backed up over using specified editing characters. The user can erase a character at a time, the whole line, or, in Interlisp-D, a word at a time. The keys that perform these editing functions are assignable via the `SETSYNTAX` function (page 6.34), with the intial settings chosen to be those most natural for the given operating system: characters are deleted one at a time by control-A under Tenex, Delete under Tops20, and BackSpace in Interlisp-D; the whole line is erased by control-Q under Tenex and in Interlisp-D, and control-U under Tops20; words are erased by control-W in Interlisp-D.

The character-deleting action on normal terminals is to echo a \ followed by the erased character; on the Interlisp-D display the character is physically erased from the screen (this action can also be specified for display terminals in other Interlisps; see page 6.43). The line-deleting action is normally to print ## and start over on a new line. Neither will back up beyond the previous carriage-return.

When reading from a file, and an end of file is encountered, all input functions close the file and generate an error, `END OF FILE` (unless `WHENCLOSE` has been used to alter this behavior; see page 6.11).

`(READ` *FILE RDTBL FLG*`)` [Function]

Reads one expression from *FILE*. Atoms are delimited by the break and separator characters as defined in *RDTBL*. To include a break or separator character in an atom, the character must be preceded by the input escape character %, e.g., `AB%(C` is the atom `AB(C`, `%%` is the atom `%`, `%`*control-A* is the atom control-A. For input from the terminal, an atom containing an interrupt character can be input by typing instead the corresponding alphabetic character preceded by control-V, e.g., `↑VC` for control-C.

Strings are delimited by double quotes. To input a string containing a double quote or a %, precede it by %, e.g., `"AB%"C"` is the string `AB"C`. Note that % can always be typed even if next character is not "special", e.g., `%A%B%C` is read as `ABC`.

If an atom is interpretable as a number, `READ` creates a number, e.g., `1E3` reads as a floating point number, `1D3` as a literal atom, `1.0` as a number, `1,0` as a literal atom, etc. An integer can be input in octal by terminating it with a `Q`, e.g., `17Q` and `15` read in as the same integer. The setting of `RADIX` (page 6.19) determines in which base integers are printed.

When reading from the terminal, all input is line-buffered to enable the action of the backspacing control characters (unless inhibited by `(CONTROL T)` (page 6.45)). Thus no characters are actually seen by the program until a carriage-return is typed.[4] However, for reading by `READ`, when a matching right parenthesis is encountered, the effect is the same as though a carriage-return were typed, i.e., the

[4] Actually, the line buffering is terminated by the character with terminal syntax class `EOL` (see page 6.33), which in most cases is carriage-return.

characters are transmitted.[5] To indicate this, Interlisp also prints a carriage-return line-feed on the terminal.

In Interlisp-10, the character control-W is defined as an IMMEDIATE read macro that erases the last expression read, echoing a \\ and the erased expression, e.g., (NOW IS THE TIME↑W \\ TIME) returns (NOW IS THE). Control-W can be used repeatedly, and can also back up and erase expressions on previous lines. However, since control-W is implemented as an IMMEDIATE read-macro character, (page 6.36), once it is typed, then individual *characters* typed before it cannot be deleted by control-A or control-Q, since they will already have passed through the line buffer.

In Interlisp-D, control-W is instead defined as an editing character that deletes the last "word" of input, i.e., back to the first non-OTHER character preceding the first non-SEPR character, essentially a repeated BackSpace. The character performing this function is assignable using the WORDDELETE syntax (page 6.34).

FLG=T suppresses the carriage-return normally typed by READ following a matching right parenthesis. (However, the characters are still given to READ; i.e., the user does not have to type the carriage-return.)

(RATOM *FILE RDTBL*) [Function]

Reads in one atom from *FILE*. Separation of atoms is defined by *RDTBL*. % is also an escape character for RATOM, and the remarks concerning line-buffering and editing control characters also apply.

If the characters comprising the atom would normally be interpreted as a number by READ, that number is returned by RATOM. Note however that RATOM takes no special action for " whether or not it is a break character, i.e., RATOM never makes a string.

(RSTRING *FILE RDTBL*) [Function]

Reads characters from *FILE* up to, but not including, the next break or separator character, and returns them as a string. Control-A, control-Q, control-V, and % have the same effect as with READ.

Note that the break or separator character that terminates a call to RATOM or RSTRING is *not* read by that call, but remains in the buffer to become the first character seen by the next reading function that is called. If that function is RSTRING, it will return the null string. This is a common source of program bugs.

(RATOMS *A FILE RDTBL*) [Function]

Calls RATOM repeatedly until the atom *A* is read. Returns a list of the atoms read, not including *A*.

(RATEST *FLG*) [Function]

If *FLG* = T, RATEST returns T if a separator was encountered immediately prior to the last atom read by RATOM, NIL otherwise.

[5] The line buffer is also transmitted to READ whenever an IMMEDIATE read-macro character is typed (page 6.36).

If *FLG* = NIL, RATEST returns T if last atom read by RATOM or READ was a break character, NIL otherwise.

If *FLG* = 1, RATEST returns T if last atom read (by READ or RATOM) contained a % (as an escape character, e.g., %[or %A%B%C), NIL otherwise.

(READC *FILE RDTBL*) [Function]

Reads and returns the next character, including %, ", etc, i.e., is not affected by break, separator, or escape character. The action of READC is subject to line-buffering, i.e., READC does not return a value until the line has been terminated even if a character has been typed. Thus, the editing control characters have their usual effect. *RDTBL* does not directly affect the value returned, but is used as usual in line-buffering, e.g., determining when input has been terminated. If (CONTROL T) has been executed (page 6.45), defeating line-buffering, the *RDTBL* argument is irrelevant, and READC returns a value as soon as a character is typed (even if the character typed is one of the editing characters, which ordinarily would never be seen in the input buffer).

(PEEKC *FILE RDTBL*) [Function]

Returns the next character, but does not actually read it and remove it from the buffer. If *RDTBL*=NIL, PEEKC is not subject to line-buffering,[6] i.e., it returns a value as soon as a character has been typed. Otherwise, PEEKC waits until the line has been terminated before returning its value. This means that control-A, control-Q, and control-V will be able to perform their usual editing functions.

(LASTC *FILE*) [Function]

Returns the last character read from *FILE*.

READ, RATOM, RATOMS, PEEKC, READC all wait for input if there is none. The only way to test whether or not there is input is to use READP:

(READP *FILE FLG*) [Function]

Returns T if there is anything in the input buffer of *FILE*, NIL otherwise. Note that because of line-buffering, READP may return T, indicating there is input in the buffer, but READ may still have to wait.

Frequently, the terminal's input buffer contains a single EOL character left over from a previous input. For most applications, this situation wants to be treated as though the buffer were empty, and so (READP T) returns NIL in this case. However, if *FLG*=T, READP also returns T in this case, i.e., (READP T T) returns T if there is *any* character in the input buffer.

(WAITFORINPUT *FILE*) [Function]

Waits until input is available from *FILE* or from the terminal, i.e. from T. WAITFORINPUT is functionally equivalent to (until (OR (READP T) (READP

[6] If reading from the terminal, the character is echoed as soon as PEEKC reads it, even though it is then "put back" into the system buffer, where a subsequent del (or control-Z on TOPS-20) before the character is read can clear it, and where subsequent line buffer backspacing could change it. Thus it is possible for the value returned by PEEKC to "disagree" in the first character with a subsequent READ.

`FILE)) do NIL)`, except that it does not use up machine cycles while waiting. Returns the device for which input is now available, i.e. *FILE* or `T`.

FILE can also be an integer, in which case `WAITFORINPUT` waits until there is input available from the terminal, or until *FILE* milliseconds have elapsed. Value is `T` if input is now available, `NIL` in the case that `WAITFORINPUT` timed out.

In Interlisp-10, `WAITFORINPUT` operates by dismissing, checking for available input, and then, if there is none, dismissing again, each time for an increasingly larger interval. The initial interval is `DISMISSINIT` milliseconds (initially 500), and the interval grows by 1/16 for each dismissal, up to a maximum of `DISMISSMAX` milliseconds (initially 10,000).

`(SKREAD` *FILE REREADSTRING*`)` [Function]

"Skip Read". It moves the file pointer for *FILE* ahead as if one call to `READ` had been performed, without paying the storage and compute cost to really read in the structure. *REREADSTRING* is for the case where the user has already performed some `READC`'s and `RATOM`'s before deciding to skip this expression. In this case, *REREADSTRING* should be the material already read (as a string), and `SKREAD` operates as though it had seen that material first, thus getting its paren-count, double-quote count, etc. set up properly.

`SKREAD` always uses `FILERDTBL` for its readtable. `SKREAD` may have difficulties if unusual read-macros have been added to `FILERDTBL`. `SKREAD` will not recognize read-macro characters in *REREADSTRING*, nor `SPLICE` or `INFIX` read macros. This is only a problem if the read-macros are defined to parse subsequent input in the file which does not follow the normal parenthesis and string-quote conventions in `FILERDTBL`.

`SKREAD` returns `%)` if the read terminated on an unbalanced closing parenthesis; `%]` if the read terminated on an unbalanced `%]`, i.e., one which also would have closed any extant open left parentheses; otherwise `NIL`.

6.3 OUTPUT FUNCTIONS

Most of the functions described below have an argument *FILE*, which specifies the name of the file on which the operation is to take place. If *FILE* is `NIL`, the primary output file is used. Some of the functions have a *RDTBL* argument, which specifies the readtable to be used for output. If *RDTBL* is `NIL`, the primary readtable is used.

Unless otherwise specified by `DEFPRINT` (page 6.23), pointers other than lists, strings, atoms, or numbers, are printed in the form {*DATATYPE*} followed by the octal representation of the address of the pointer (regardless of radix). For example, an array pointer might print as `{ARRAYP}#43,2760`. This printed representation is for compactness of display on the user's terminal, and will *not* read back in correctly; if the form above is read, it will produce the atom "`{ARRAYP}#43,2760`".

Note: the term *end-of-line* appearing in the description of an output function means the character or characters used to terminate a line in the file system being used by the given implementation of Interlisp. For example, in Interlisp-10 end-of-line is indicated by the characters carriage-return and line-feed in that order.

(PRIN1 *X FILE*) [Function]

Prints *X* on *FILE*.

(PRIN2 *X FILE RDTBL*) [Function]

Prints *X* on *FILE* with %'s and "'s inserted where required for it to read back in properly by READ, using *RDTBL*.

Both PRIN1 and PRIN2 print lists as well as atoms and strings; PRIN1 is usually used only for explicitly printing formatting characters, e.g., (PRIN1 (QUOTE %[)) might be used to print a left square bracket (the % would not be printed by PRIN1). PRIN2 is used for printing S-expressions which can then be read back into Interlisp with READ; i.e., break and separator characters in atoms will be preceded by %'s. For example, the atom "()" is printed as %(%) by PRIN2. If RADIX=8 (page 6.19), PRIN2 prints a Q after integers but PRIN1 does not (but both print the integer in octal).

(PRIN3 *X FILE*) [Function]
(PRIN4 *X FILE RDTBL*) [Function]

PRIN3 and PRIN4 are the same as PRIN1 and PRIN2 respectively, except that they do not increment the horizontal position counter nor perform any linelength checks. They are useful primarily for printing control characters.

(PRINT *X FILE RDTBL*) [Function]

Prints the expression *X* using PRIN2 followed by an end-of-line. Returns *X*.

(SPACES *N FILE*) [Function]

Prints *N* spaces. Returns NIL.

(TERPRI *FILE*) [Function]

Prints an end-of-line. Returns NIL.

(TAB *POS MINSPACES FILE*) [Function]

Prints the appropriate number of spaces to move to position *POS*. *MINSPACES* indicates how many spaces must be printed (if NIL, 1 is used). If the current position plus *MINSPACES* is greater than *POS*, TAB does a TERPRI and then (SPACES *POS*). If *MINSPACES* is T, and the current position is greater than *POS*, then TAB does nothing.

Note: A sequence of PRINT, PRIN2, SPACES, and TERPRI expressions can often be more conveniently coded with a single PRINTOUT statement (page 6.25).

(SHOWPRIN2 *X FILE RDTBL*) [Function]

Like PRIN2 except if SYSPRETTYFLG=T, prettyprints *X* instead. Returns *X*.

(SHOWPRINT *X FILE RDTBL*) [Function]

Like PRINT except if SYSPRETTYFLG=T, prettyprints *X* instead, followed by an end-of-line. Returns *X*.

SHOWPRINT and SHOWPRIN2 are used by the programmer's assistant (page 8.1) for printing the values of expressions and for printing the history list, by various commands of the break package (page 9.1), e.g. ?= and BT commands, and various other system packages. The idea is that by simply settting or binding SYSPRETTYFLG to T (initially NIL), the user instructs the system when interacting with the user to PRETTYPRINT expressions (page 6.47) instead of printing them.

(PRINTBELLS) [Function]

Used by DWIM (page 15.1) to print a sequence of bells to alert the user to stop typing. Can be advised or redefined for special applications, e.g., to flash the screen on a display terminal.

(DOBE) [Function]

(Interlisp-10) Dismiss until Output Buffer is Empty, i.e., until all of the characters that have been printed by Interlisp functions have actually been printed on the user's terminal. For example, it is important to perform a DOBE after printing an error message before clearing the input buffers to make sure that the user has actually seen the error message.

In systems that do not handle output to the display asynchronously with user computation, such as Interlisp-D, DOBE is a no-op.

6.3.1 Printlevel

When using Interlisp one often has to handle large, complicated lists, which are difficult to understand when printed out. PRINTLEVEL allows the user to specify in how much detail lists should be printed. The print functions PRINT, PRIN1, and PRIN2 are all affected by level parameters set by:

(PRINTLEVEL *CARVAL* *CDRVAL*) [Function]

Sets the CAR print level to *CARVAL*, and the CDR print level to *CDRVAL*. Returns a list cell whose CAR and CDR are the old settings. PRINTLEVEL is initialized with the value (1000 . -1).

In order that PRINTLEVEL can be used with RESETFORM or RESETSAVE, if *CARVAL* is a list cell it is equivalent to (PRINTLEVEL (CAR *CARVAL*) (CDR *CARVAL*)).

(PRINTLEVEL *N* NIL) changes the CAR printlevel without affecting the CDR printlevel. (PRINTLEVEL NIL *N*) changes the CDR printlevel with affecting the CAR printlevel. (PRINTLEVEL) gives the current setting without changing either.

The CAR printlevel specifies how "deep" to print a list. Specifically, it is the number of unpaired left parentheses which will be printed. Below that level, all lists will be printed as &. For example, suppose *X* = (A (B C (D (E F) G) H) K). If *CARVAL*=3, (PRINT *X*) would print (A (B C (D & G) H) K), if *CARVAL*=2, (A (B C & H) K), if *CARVAL*=1, (A & K), and if *CARVAL*=0, just &.

If the CAR printlevel is *negative*, the action is similar except that an end-of-line is inserted after each right parentheses that would be immediately followed by a left parenthesis.

The CDR printlevel specifies how "long" to print a list. It is the number of top level list elements that will be printed before the printing is terminated with --. For example, if *CDRVAL*=2, (A B C D E) will print as (A B --). For sublists, the number of list elements printed is also affected by the depth of printing in the CAR direction: Whenever the *sum* of the depth of the sublist (i.e. the number of unmatched left parentheses) and the number of elements is greater than the CDR printlevel, -- is printed. This gives a "triangular" effect in that less is printed the farther one goes in either CAR or CDR direction. For example, if *CDRVAL*=2, then (A (B C (D (E F) G) H) K L) will print as (A (B --) --)

and if *CDRVAL*=3, as `(A (B C --) K --)`.

If the CDR printlevel is negative, then it is the same as if the CDR printlevel were infinite.

The printlevel setting can be changed dynamically, even while Interlisp is printing, by typing control-P followed by a number, i.e., a string of digits, followed by a period or exclamation point. As soon as control-P is typed, Interlisp clears and saves the input buffer, clears the output buffer, rings the bell indicating it has seen the control-P, and then waits for input, which is terminated by any non-number. The input buffer is then restored and the program continues. If the input was terminated by a period or an exclamation point, the CAR printlevel is immediately set to this number; otherwise, the input is ignored. Characters cleared from the output buffer will have been lost in either case, and printing continues with the (possibly new) printlevel. If the print routine is currently deeper than the new level, all unfinished lists above that level will be terminated by "--)". Thus, if a circular or long list of atoms, is being printed out, typing "control-P0." will cause the list to be terminated immediately.

If the string of digits following a control-P is terminated by a comma, another number may be typed terminated by a period or exclamation point. The CAR printlevel will then be set to the first number, the CDR printlevel to the second number.

In either case, if a period is used to terminate the printlevel setting, the printlevel will be returned to its previous setting after the current printout has finished. If an exclamation point is used, the change is ·permanent and the printlevel is not restored (until it is changed again).

PLVLFILEFLG [Variable]

Normally, PRINTLEVEL only affects terminal output. Output to all other files acts as though the print level is infinite. However, if PLVLFILEFLG is T (initially NIL), then PRINTLEVEL affects output to files as well.

6.3.2 Printing numbers

How the ordinary printing functions (PRIN1, PRIN2, etc.) print numbers can be affected in several ways. RADIX influences the printing of integers, and FLTFMT influences the printing of floating point numbers. The setting of the variable PRXFLG determines how the symbol-manipulation functions handle numbers. The PRINTNUM package permits greater controls on the printed appearance of numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

(RADIX *N*) [Function]

Resets the output radix for integers to the absolute value of *N*. If *N* is negative, integers are interpreted by the print routines as unsigned numbers; i.e., the actual two's complement representation of the integer in the integer size of the particular implementation is interpreted as if it were a positive number on a machine of infinite integer size. Thus, numeric output under a negative radix varies with the implementation, and numbers printed in this way by one implementation will not read correctly in an implementation whose integers are of a different size.

For example, in Interlisp-10, whose integer size is 36 bits, -9 will print as shown with the following radices:

(RADIX)	(PRINT -9)
10	-9
8	-11Q
-10	68719476727 (i.e. 2^{36}-9)
-8	777777777767Q

The value of RADIX is its previous setting. (RADIX) gives the current setting without changing it. The initial setting is 10.

Note that RADIX affects output *only*. There is no input radix; on input, numbers are interpreted as decimal unless they end in Q, in which case they are interpreted as octal. Thus READ and PRINT are inverses, independent of any radix setting. RADIX also does not affect the behavior of UNPACK, etc., unless the value of PRXFLG (below) is T; e.g., with (RADIX 8), the value of (UNPACK 9) is (9), not (1 1).

(FLTFMT *FORMAT*) [Function]

Resets the output format for floating point numbers to the FLOAT format *FORMAT* (see PRINTNUM below for a description of FLOAT formats). *FORMAT*=T specifies the default "free" formatting: some number of significant digits (a function of the implementation) are printed, with trailing zeros suppressed; numbers with sufficiently large or small exponents are instead printed in exponent notation.

FLTFMT returns its current setting. (FLTFMT) returns the current setting without changing it. The initial setting is T.

In Interlisp-10, *FORMAT* may also be a machine-dependent FLOAT format-code as returned by NUMFORMATCODE (page 6.23).

Whether print name manipulation functions (UNPACK, NCHARS, etc.) use the values of RADIX and FLTFMT is determined by the variable PRXFLG:

PRXFLG [Variable]

If PRXFLG=NIL (the initial setting), then the "PRIN1" name used by PACK, UNPACK, MKSTRING, etc., is computed using base 10 for integers and the system default floating format for floating point numbers, independent of the current setting of RADIX or FLTFMT. If PRXFLG=T, then RADIX and FLTFMT do dictate the "PRIN1" name of numbers. Note that in this case, PACK and UNPACK are *not* inverses.

Examples with (RADIX 8), (FLTFMT '(FLOAT 4 2)):

With PRXFLG=NIL,

```
(UNPACK 13)  =>  (1 3)

(PACK '(A 9))  =>  A9
```

```
(UNPACK 1.2345)  =>  (1 %. 2 3 4 5)
```

With PRXFLG=T,

```
(UNPACK 13)  =>  (1 5)

(PACK '(A 9))  =>  A11

(UNPACK 1.2345)  =>  (1 %. 2 3)
```

Note that PRXFLG does not effect the radix of "PRIN2" names, so with (RADIX 8), (NCHARS 9 T), which uses PRIN2 names, would return 3, (since 9 would print as 11Q) for either setting of PRXFLG.

Warning: Some system functions will not work correctly if PRXFLG is not NIL. Therefore, resetting the global value of PRXFLG is not recommended. It is much better to rebind PRXFLG as a SPECVAR for that part of a program where it needs to be non-NIL.

The basic function for printing numbers under format control is PRINTNUM. Its utility is considerably enhanced when used in conjunction with the PRINTOUT package (page X.XX), which implements a compact language for specifying complicated sequences of elementary printing operations, and makes fancy output formats easy to design and simple to program.

(PRINTNUM *FORMAT NUMBER FILE*) [Function]

Prints *NUMBER* on *FILE* according to the format *FORMAT*. *FORMAT* is a list structure with one of the forms described below. *FORMAT* can also be a machine dependent format-code as returned by NUMFORMATCODE (page 6.23).

(Interlisp-10) If *NUMBER* does not fit in the field specified by *FORMAT*, the full print name is printed. Then a TAB is executed so that the line position of the file after PRINTNUM is always the position prior to printing plus the indicated width.

If *FORMAT* is a list of the form (FIX *WIDTH RADIX PAD0 LEFTFLUSH*), this specifies a FIX format. *NUMBER* is rounded to the nearest integer, and then printed in a field *WIDTH* characters long with radix set to *RADIX* (or 10 if *RADIX*=NIL; note that the setting of RADIX is *not* used as the default). If *PAD0* and *LEFTFLUSH* are both NIL, the number is right-justified in the field, and the padding characters to the left of the leading digit are spaces. If *PAD0* is T, the character "0" is used for padding. If *LEFTFLUSH* is T, then the number is left-justified in the field, with trailing spaces to fill out *WIDTH* characters.

The following examples illustrate the effects of the FIX format options (the vertical bars indicate the field width):

```
FORMAT               NUMBER          PRINTNUM prints

(FIX 2)              3               | 3|

(FIX 2 NIL T)        7               |07|

(FIX 12 8 T)         14              |000000000016|

(FIX 5 NIL NIL T)    2               |2    |
```

If *FORMAT* is a list of the form (FLOAT *WIDTH DECPART EXPPART PAD0 ROUND*), this specifies a FLOAT format. *NUMBER* is printed as a decimal number in a field *WIDTH* characters wide, with *DECPART* digits to the right of the decimal point. If *EXPPART* is not 0 (or NIL), the number is printed in exponent notation, with the exponent occupying *EXPPART* characters in the field. *EXPPART* should allow for the character E and an optional sign to be printed before the exponent digits. As with FIX format, padding on the left is with spaces, unless *PAD0* is T. If *ROUND* is given, it indicates the digit position at which rounding is to take place, counting from the leading digit of the number.[7]

FLOAT format examples:

```
FORMAT                   NUMBER      PRINTNUM prints

(FLOAT 7 2)              27.689      |  27.69|

(FLOAT 7 2 NIL T)        27.689      |0027.69|

(FLOAT 7 2 2)            27.689      | 2.77E1|

(FLOAT 11 2 4)           27.689      |   2.77E+01|[8]

(FLOAT 7 2 NIL NIL 1)    27.689      |  30.00|

(FLOAT 7 2 NIL NIL 2)    27.689      |  28.00|
```

NILNUMPRINTFLG [Variable]

If PRINTNUM's *NUMBER* argument is not a number and not NIL, a NON-NUMERIC ARG error is generated. If *NUMBER* is NIL, the effect depends on the setting of the variable NILNUMPRINTFLG. If NILNUMPRINTFLG is NIL, then the error occurs as usual. If it is non-NIL, then no error occurs, and the value of NILNUMPRINTFLG is printed right-justified in the field described by *FORMAT*. This option facilitates the printing of numbers in aggregates with missing values coded as NIL.

[7] The interpretation of *WIDTH*=NIL and *DECPART*=NIL are not specified, and are currently a function of the implementation. Interlisp-10 prohibits *WIDTH*=NIL, and treats *DECPART*=NIL as equivalent to *DECPART*=0; Interlisp-D interprets *WIDTH*=NIL to mean no padding, i.e., to use however much space the number needs, and interprets *DECPART*=NIL to mean as many decimal places as needed.

[8] As of this writing, the Interlisp-10 implementation actually does something less intuitive with the *EXPPART* field: the placement of the decimal point is affected by *DECPART*, and padding never occurs. These two examples in Interlisp-10 would actually print as |.28E+02| and |27.69E+0000|.

In some implementations, formatted printing of numbers receives assistance from the operating system, provided that the format is specified in some sort of special code. PRINTNUM works by converting the machine-independent format specifications described above into machine-*dependent* codes the exact form of which may vary from implementation to implementation. This conversion process takes place on each call to PRINTNUM. For efficiency purposes, if the user is going to be performing a particular call to PRINTNUM frequently, he may wish to separate the conversion from the actual printing, performing the conversion process just once and saving the result. The function NUMFORMATCODE is available for this purpose: NUMFORMATCODE takes a format, performs the conversion and returns a machine dependent format-code, which can be given to PRINTNUM in place of a list structure format as described above. In this case, PRINTNUM will not have to perform the conversion, but can simply use the machine-dependent format code directly.

(NUMFORMATCODE *FORMAT SMASHCODE*) [Function]

Converts the FIX or FLOAT format *FORMAT* to a machine-dependent format-code. If *SMASHCODE* is recognized as a format-code data-structure, then the new format-code is smashed into that structure instead of allocating new storage. (NUMFORMATCODE) returns an uninitialized datum that can later be smashed.

In Interlisp-D, this function is a no-op, as there is no special internal representation for number formats.

6.3.3 User Defined Printing

(DEFPRINT *TYPE FN*) [Function]

TYPE is a type name (see page 2.1). Whenever a printing function (PRINT, PRIN1, PRIN2, etc.) encounters an object of the indicated type, *FN* is called with the item to be printed as its argument. If it returns NIL, the datum is printed in the manner the system defaults; for user data types, it is printed as {datatype}#nnnnnn. If *FN* wishes to specify how the datum should be printed, it should return a list of the form (*ITEM1* . *ITEM2*). *ITEM1* is printed using PRIN1 (unless it is NIL), and then *ITEM2* printed using PRIN2 with no spaces between the two items. (Typically, *ITEM1* is a read macro character.)

In Interlisp-10, *TYPE* may also be a type number (see page 22.2). Note that the user can specify different action for type names ARRAYP, HARRAYP, TERMTABLEP, READTABLEP, and CCODEP, even though they all have the same type *number*.

Note that DEFPRINT also affects internal calls to print from PACK, CONCAT, etc., i.e. any operation that involves obtaining a print name (see page 2.8). A consequence of this fact is that in implementations that do not have reentrant printing code (in particular, Interlisp-10), the user's DEFPRINT function must *not* call any print name manipulating functions itself, or the results of the whole printing operation are undefined.

6.3.4 Dumping Unusual Data Structures

HPRINT (for "Horrible Print") and HREAD provide a mechanism for printing and reading back in general data structures that cannot normally be dumped and loaded easily, such as (possibly re-entrant or circular) structures containing user datatypes, arrays, hash tables, as well as list structures. HPRINT will correctly print and read back in any structure containing any or all of the above, chasing all pointers down to the level of literal atoms, numbers or strings. HPRINT currently cannot handle compiled code arrays, stack positions, or arbitrary unboxed numbers.

HPRINT operates by simulating the Interlisp PRINT routine for normal list structures. When it encounters a user datatype (see page 3.14), or an array or hash array, it prints the data contained therein, surrounded by special characters defined as read-macro characters (see page 6.36). While chasing the pointers of a structure, it also keeps a hash table of those items it encounters, and if any item is encountered a second time, another read-macro character is inserted before the first occurrence (by resetting the file pointer with SETFILEPTR) and all subsequent occurrences are printed as a back reference using an appropriate macro character. Thus the inverse function, HREAD merely calls the Interlisp READ routine with the appropriate readtable.

(HPRINT *EXPR FILE UNCIRCULAR DATATYPESEEN*) [Function]

Prints *EXPR* on *FILE*. If *UNCIRCULAR* is non-NIL, HPRINT does no checking for any circularities in *EXPR* (but is still useful for dumping arbitrary structures of arrays, hash arrays, lists, user data types, etc., that do not contain circularities). Specifying *UNCIRCULAR* as non-NIL results in a large speed and internal-storage advantage.

Normally, when HPRINT encounters a user data type for the first time, it outputs a summary of the data type's declaration. When this is read in, the data type is redeclared. If *DATATYPESEEN* is non-NIL, HPRINT will assume that the same data type declarations will be in force at read time as were at HPRINT time, and not output declarations.

HPRINT is intended primarily for output to disk files, since the algorithm depends on being able to reset the file pointer. If *FILE* is not a disk file (and *UNCIRCULAR* = NIL), a temporary file, HPRINT.SCRATCH, is opened, *EXPR* is HPRINTed on it, and then that file is copied to the final output file and the temporary file is deleted.

(HREAD *FILE*) [Function]

Reads and returns an HPRINT-ed expression from *FILE*.

(HCOPYALL *X*) [Function]

Copies data structure *X*. *X* may contain circular pointers as well as arbitrary structures.

Note: HORRIBLEVARS and UGLYVARS (page 11.25) are two file package commands for dumping and reloading circular and re-entrant data structures. They provide a convenient interface to HPRINT and HREAD.

6.4 READFILE AND WRITEFILE

For those applications where the user simply wants to simply read all of the expressions on a file, and not evaluate them, the function READFILE is available:

(READFILE *FILE*) [Function]

Reads successive expressions from file using READ (with FILERDTBL as readtable) until the single atom STOP is read, or an end of file encountered. Returns a list of these expressions.

(WRITEFILE *X* *FILE*) [Function]

Inverse of READFILE. Writes a date expression onto *FILE*, followed by successive expressions from *X*, using FILERDTBL as a readtable. If *X* is atomic, its value is used. If *FILE* is not open, it is opened. If *FILE* is a list, (CAR *FILE*) is used and the file is left opened. Otherwise, when *X* is finished, a STOP is printed on *FILE* and it is closed. Returns *FILE*.

(ENDFILE *FILE*) [Function]

Prints STOP on *FILE* and closes it.

6.5 PRINTOUT

Interlisp provides many facilities for controlling the format of printed output. By executing various sequences of PRIN1, PRIN2, TAB, TERPRI, SPACES, PRINTNUM, and PRINTDEF, almost any effect can be achieved. PRINTOUT implements a compact language for specifying complicated sequences of these elementary printing functions. It makes fancy output formats easy to design and simple to program.

PRINTOUT is a CLISP word (like for and if) for interpreting a special printing language in which the user can describe the kinds of printing desired. The description is translated by DWIMIFY to the appropriate sequence of PRIN1, TAB, etc., before it is evaluated or compiled. PRINTOUT printing descriptions have the following general form:

(PRINTOUT *FILE* *PRINTCOM*$_1$ *PRINTCOM*$_2$ ··· *PRINTCOM*$_N$)

FILE is evaluated to obtain the name of the file to which the output from this specification is directed. The PRINTOUT commands are strung together, one after the other without punctuation, after *FILE*. Some commands occupy a single position in this list, but many commands expect to find arguments following the command name in the list. The commands fall into several logical groups: one set deals with horizontal and vertical spacing, another group provides controls for certain formatting capabilities (font changes and subscripting), while a third set is concerned with various ways of actually printing items. Finally, there is a command that permits escaping to a simple Lisp evaluation in the middle of a PRINTOUT form. The various commands are described below. The following examples give a general flavor of how PRINTOUT is used:

Example 1: Suppose the user wanted to print out on the terminal the values of three variables, X, Y, and Z, separated by spaces and followed by a carriage return. This could be done by:

```
(PRIN1 X T)
(SPACES 1 T)
(PRIN1 Y T)
(SPACES 1 T)
(PRIN1 Z T)
(TERPRI T)
```

or by the more concise PRINTOUT form:

```
(PRINTOUT T X , Y , Z T)
```

Here the first T specifies output to the terminal, the commas cause single spaces to be printed, and the final T specifies a TERPRI. The variable names are not recognized as special PRINTOUT commands, so they are printed using PRIN1 by default.

Example 2: Suppose the values of X and Y are to be pretty-printed lined up at position 10, preceded by identifying strings. If the output is to go to the primary output file, the user could write either:

```
(PRIN1 "X =")
(PRINTDEF X 10 T)
(TERPRI )
(PRIN1 "Y =")
(PRINTDEF Y 10 T)
(TERPRI)
```

or the equivalent:

```
(PRINTOUT NIL "X =" 10 .PPV X T "Y =" 10 .PPV Y T)
```

Since strings are not recognized as special commands, "X =" is also printed with PRIN1 by default. The positive integer means TAB to position 10, where the .PPV command causes the value of X to be prettyprinted as a variable. By convention, special atoms used as PRINTOUT commands are prefixed with a period. The T causes a carriage return, so the Y information is printed on the next line.

Example 3. As a final example, suppose that the value of X is an integer and the value of Y is a floating-point number. X is to be printed right-flushed in a field of width 5 beginning at position 15, and Y is to be printed in a field of width 10 also starting at position 15 with 2 places to the right of the decimal point. Furthermore, suppose that the variable names are to appear in the font named BOLDFONT and the values in font SMALLFONT. The program in ordinary Lisp that would accomplish these effects is too complicated to include here. With PRINTOUT, one could write:

```
(PRINTOUT NIL
      .FONT BOLDFONT "X =" 15
      .FONT SMALLFONT .I5 X T
      .FONT BOLDFONT "Y =" 15
      .FONT SMALLFONT .F10.2 Y T
      .FONT BOLDFONT)
```

The .FONT commands do whatever is necessary to change the font on a multi-font output device. The .I5 command sets up a FIX format for a call to the function PRINTNUM (page 6.21) to print X in the desired format. The .F10.2 specifies a FLOAT format for PRINTNUM.

6.5.1 Horizontal Spacing Commands

The horizontal spacing commands provide convenient ways of calling TAB and SPACES. In the following descriptions, *N* stands for a literal positive integer.

N Used for absolute spacing. It results in a TAB to position *N* (literally, a (TAB *N*)). If the line is currently at position *N* or beyond, the file will be positioned at position *N* on the next line.

`.TAB` *POS* — Specifies `TAB` to position (the value of) *POS*. This is one of several commands whose effect could be achieved by simply escaping to Lisp, and executing the corresponding form. It is provided as a separate command so that the `PRINTOUT` form is more concise and is prettyprinted more compactly. Note that `.TAB` *N* and *N*, where *N* is an integer, are equivalent.

`.TAB0` *POS* — Like `.TAB` except that it can result in zero spaces (i.e. the call to `TAB` specifies *MINSPACES*=0).

`-`*N* — Negative integers indicate relative (as opposed to absolute) spacing. Translates as `(SPACES |`*N*`|)`.

`, ,, ,,,` — Provides a short-hand way of specifying 1, 2 or 3 spaces, i.e., these commands are equivalent to `-1`, `-2`, and `-3`, respectively.

`.SP` *DISTANCE* — Translates as `(SPACES` *DISTANCE*`)`. Note that `.SP` *N* and `-`*N*, where *N* is an integer, are equivalent.

`.RESET` — Resets the current line by causing a carriage-return to be printed without a line-feed. Useful for overprinting, or for regaining control of a line on which characters have been printed in a variable pitched font.

6.5.2 Vertical Spacing Commands

Vertical spacing is obtained by calling `TERPRI` or printing form-feeds. The relevant commands are:

`T` — Translates as `(TERPRI)`. This command is functionally equivalent to the integer command `0`; they both move to position 0 (= column 1) of the next line. To print the letter `T`, use the string `"T"`.

`.SKIP` *LINES* — Equivalent to a sequence of *LINES* `(TERPRI)`'s. The `.SKIP` command allows for skipping large constant distances and for computing the distance to be skipped.

`.PAGE` — Puts a form-feed (control-L) out on the file. Care is taken to make sure that Interlisp's view of the current line position is correctly updated.

6.5.3 Special Formatting Controls

There are a small number of commands for invoking some of the formatting capabilities of multi-font output devices. The available commands are:

`.FONT` *FONTSPEC* — Puts out a control sequence that causes a change to font *FONTSPEC* (the association between *FONTSPEC* and a specific font must be defined in the user's font profile, as described in page 6.55). *FONTSPEC* may be a font-name variable or an expression that evaluates to the value of a font-name variable. *FONTSPEC* may also be a positive integer *N*, which is taken as an abbreviated reference to the font named `FONT`*N* (e.g. 1 => `FONT1`).

`.SUP` — Specifies superscripting. All subsequent characters are printed above the base of the current line. Note that this is absolute, not relative: a `.SUP` following a `.SUP` is a no-op.

`.SUB` Specifies subscripting. Subsequent printing is below the base of the current line. As with superscripting, the effect is absolute.

`.BASE` Moves printing back to the base of the current line. Un-does a previous `.SUP` or `.SUB`; a no-op, if printing is currently at the base.

6.5.4 Printing Specifications

The value of any expression in a `PRINTOUT` form that is not recognized as a command itself or as a command argument is printed using `PRIN1` by default. For example, title strings can be printed by simply including the string as a separate `PRINTOUT` command, and the values of variables and forms can be printed in much the same way. Note that a literal integer, say 51, cannot be printed by including it as a command, since it would be interpreted as a `TAB`; the desired effect can be obtained by using instead the string specification "51", or the form `(QUOTE 51)`.

For those instances when `PRIN1` is not appropriate, e.g., `PRIN2` is required, or a list structures must be prettyprinted, the following commands are available:

`.P2` *THING* Causes *THING* to be printed using `PRIN2`; translates as `(PRIN2` *THING*`)`.

`.PPF` *THING* Causes *THING* to be prettyprinted at the current line position via `PRINTDEF` (page 6.49). The call to `PRINTDEF` specifies that *THING* is to be printed as if it were part of a function definition. That is, `SELECTQ`, `PROG`, etc., receive special treatment.

`.PPV` *THING* Prettyprints *THING* as a variable; no special interpretation is given to `SELECTQ`, `PROG`, etc.

`.PPFTL` *THING* Like `.PPF`, but prettyprints *THING* as a *tail*, that is, without the initial and final parentheses if it is a list. Useful for prettyprinting sub-lists of a list whose other elements are formatted with other commands.

`.PPVTL` *THING* Like `.PPV`, but prettyprints *THING* as a tail.

6.5.4.1 Paragraph Format

Interlisp's prettyprint routines are designed to display the structure of expressions, but they are not really suitable for formatting unstructured text. If a list is to be printed as a textual paragraph, its internal structure is less important than controlling its left and right margins, and the indentation of its first line. The `.PARA` and `.PARA2` commands allow these parameters to be conveniently specified.

`.PARA` *LMARG RMARG LIST*

Prints *LIST* in paragraph format, using `PRIN1`. Translates as `(PRINTPARA` *LMARG RMARG LIST*`)` (see page 6.31). Example: `(PRINTOUT T 10 .PARA 5 -5 LST)` will print the elements of `LST` as a paragraph with left margin at 5, right margin at `(LINELENGTH)`-5, and the first line indented to 10.

`.PARA2` *LMARG RMARG LIST*

. Print as paragraph using PRIN2 instead of PRIN1. Translates as (PRINTPARA *LMARG RMARG LIST* T).

6.5.4.2 Right-Flushing

Two commands are provided for printing simple expressions flushed-right against a specified line position, using the function FLUSHRIGHT (page 6.31). They take into account the current position, the number of characters in the print-name of the expression, and the position the expression is to be flush against, and then print the appropriate number of spaces to achieve the desired effect. Note that this might entail going to a new line before printing. Note also that right-flushing of expressions longer than a line (e.g. a large list) makes little sense, and the appearance of the output is not guaranteed.

.FR *POS EXPR* Flush-right using PRIN1. The value of *POS* determines the position that the right end of *EXPR* will line up at. As with the horizontal spacing commands, a negative position number means |*POS*| columns from the current position, a positive number specifies the position absolutely. *POS*=0 specifies the right-margin, i.e. is interpreted as (LINELENGTH).

.FR2 *POS EXPR* Flush-right using PRIN2 instead of PRIN1.

6.5.4.3 Centering

Commands for centering simple expressions between the current line position and another specified position are also available. As with right flushing, centering of large expressions is not guaranteed.

.CENTER *POS EXPR*

Centers *EXPR* between the current line position and the position specified by the value of *POS*. A positive *POS* is an absolute position number, a negative *POS* specifies a position relative to the current position, and 0 indicates the right-margin. Uses PRIN1 for printing.

.CENTER2 *POS EXPR*

Centers using PRIN2 instead of PRIN1.

6.5.4.4 Numbering

The following commands provide FORTRAN-like formatting capabilities for integer and floating-point numbers. Each command specifies a printing format and a number to be printed. The format specification translates into a format-list for the function PRINTNUM (see page 6.21).

.I*FORMAT NUMBER*

Specifies integer printing. Translates as a call to the function PRINTNUM with a FIX format-list constructed from *FORMAT*. The atomic format is broken apart at internal periods to form the format-list. For example, .I5.-8.T yields the format-list (FIX 5 -8 T), and the command sequence (PRINTOUT T .I5.-8.T FOO) will translate as (PRINTNUM '(FIX 5 -8 T) FOO). It will cause the value of FOO to be printed with radix -8 right-flushed in a field of width 5, with 0's used for padding on the left. Internal NIL's may be omitted, e.g. the commands .I5..T and .I5.NIL.T are equivalent.

.F*FORMAT NUMBER*

Specifies floating-number printing. Like the .I format command, except translates with a FLOAT format-list.

.N *FORMAT NUMBER*

The .I and .F commands specify calls to PRINTNUM with quoted format specifications. The .N command translates as (PRINTNUM *FORMAT NUMBER*), i.e., it permits the format to be the value of some expression. Note that, unlike the .I and .F commands, *FORMAT* is a separate element in the command list, not part of an atom beginning with .N.

6.5.5 Escaping to LISP

There are many reasons for taking control away from PRINTOUT in the middle of a long printing expression. Common situations involve temporary changes to system printing parameters (e.g. LINELENGTH), conditional printing (e.g. print FOO only if FIE is T), or lower-level iterative printing within a higher-level print specification.

FORM

The escape command. *FORM* is an arbitrary Lisp expression that is evaluated within the context established by the PRINTOUT form, i.e., *FORM* can assume that the primary output file has been set to be the *FILE* argument to PRINTOUT. Note that nothing is done with the *value* of *FORM*; any printing desired is accomplished by *FORM* itself, and the value is discarded.

Note: Although PRINTOUT logically encloses its translation in a RESETFORM (page 9.20) to change the primary output file to the *FILE* argument (if non-NIL), in most cases it can actually pass *FILE* (or a locally bound variable if *FILE* is a non-trivial expression) to each printing function. Thus, the RESETFORM is only generated when the # command is used, or user-defined commands (below) are used. If many such occur in repeated PRINTOUT forms, it may be more efficient to embed them all in a single RESETFORM which changes the primary output file, and then specify *FILE*=NIL in the PRINTOUT expressions themselves.

6.5.6 User-Defined Commands

The collection of commands and options outlined above is aimed at fulfilling all common printing needs. However, certain applications might have other, more specialized printing idioms, so a facility is provided whereby the user can define new commands. This is done by adding entries to the global list PRINTOUTMACROS to define how the new commands are to be translated.

PRINTOUTMACROS [Variable]

PRINTOUTMACROS is an association-list whose elements are of the form (*COMM FN*). Whenever *COMM* appears in command position in the sequence of PRINTOUT commands (as opposed to an argument position of another command), *FN* is applied to the tail of the command-list (including the command).

After inspecting as much of the tail as necessary, the function must return a list whose CAR is the translation of the user-defined command and its arguments, and whose CDR is the list of commands still remaining to be translated in the normal way.

For example, suppose the user wanted to define a command "?", which will cause its single argument to be printed with PRIN1 only if it is not NIL. This can be done by entering (? ?TRAN) on PRINTOUTMACROS, and defining the function ?TRAN as follows:

```
(LAMBDA (COMS)
   (CONS (SUBST (CADR COMS) 'ARG
                '(PROG ((TEMP ARG))
                       (COND (TEMP (PRIN1 TEMP)))))
         (CDDR COMS)))
```

Note that ?TRAN does not do any printing itself; it returns a form which, when evaluated in the proper context, will perform the desired action. This form should direct all printing to the primary output file.

6.5.7 Special Printing Functions

The paragraph printing commands are translated into calls on the function PRINTPARA, which may also be called directly:

(PRINTPARA *LMARG RMARG LIST P2FLAG PARENFLAG FILE*) [Function]

Prints *LIST* on *FILE* in line-filled paragraph format with its first element beginning at the current line position and ending at or before *RMARG*, and with subsequent lines appearing between *LMARG* and *RMARG*. If *P2FLAG* is non-NIL, prints elements using PRIN2, otherwise PRIN1. If *PARENFLAG* is non-NIL, then parentheses will be printed around the elements of *LIST*.

If *LMARG* is zero or positive, it is interpreted as an absolute column position. If it is negative, then the left margin will be at |*LMARG*|+(POSITION). If *LMARG*=NIL, the left margin will be at (POSITION), and the paragraph will appear in block format.

If *RMARG* is positive, it also is an absolute column position (which may be greater than the current (LINELENGTH)). Otherwise, it is interpreted as relative to (LINELENGTH), i.e., the right margin will be at (LINELENGTH)+|*RMARG*|. Example: (TAB 10) (PRINTPARA 5 -5 LST T) will PRIN2 the elements of LST in a paragraph with the first line beginning at column 10, subsequent lines beginning at column 5, and all lines ending at or before (LINELENGTH)-5.

The current (LINELENGTH) is unaffected by PRINTPARA, and upon completion, *FILE* will be positioned immediately after the last character of the last item of *LIST*. PRINTPARA is a no-op if *LIST* is not a list.

The right-flushing and centering commands translate as calls to the function FLUSHRIGHT:

(FLUSHRIGHT *POS X MIN P2FLAG CENTERFLAG FILE*) [Function]

If *CENTERFLAG*=NIL, prints *X* right-flushed against position *POS* on *FILE*; otherwise, centers *X* between the current line position and *POS*. Makes sure that it spaces over at least *MIN* spaces before printing by doing a TERPRI if necessary; *MIN*=NIL is equivalent to *MIN*=1. A positive *POS* indicates an absolute position, while a negative *POS* signifies the position which is |*POS*| to the right of the current line position. *POS*=0 is interpreted as (LINELENGTH), the right margin.

6.6 READTABLES

Many Interlisp input functions treat certain characters in special ways. For example, READ recognizes that the right and left parenthesis characters are used to specify list structures, and that the quote character is used to delimit text strings. The Interlisp input and (to a certain extent) output routines are table driven by readtables. Readtables are objects that specify the syntactic properties of characters for input routines. Since the input routines parse character sequences into objects, the readtable in use determines which sequences are recognized as literal atoms, strings, list structures, etc.

Most Interlisp input functions take an optional readtable argument, which specifies the readtable to use when reading an expression. If NIL is given as the readtable, the "primary readtable" is used. If T is specified, the system terminal readtable is used. Some functions will also accept the atom ORIG (*not* the *value* of ORIG) as indicating the "original" system readtable. Some output functions also take a readtable argument. For example, PRIN2 prints an expression so that it would be read in correctly using a given readtable.

The Interlisp system uses three readtables: T for input/output from terminals, the value of FILERDTBL for input/output from files, and the value of EDITRDTBL for input from terminals while in the editor. These three tables are initially copies of the ORIG readtable, with changes made to some of them to provide read macros (page 6.36) that are specific to terminal input or file input. Using the functions described below, the user may further change, reset, or copy these tables. The user can also create new readtables, and either explicitly pass them to input/output functions as arguments, or install them as the primary readtable, via SETREADTABLE, and then not specify a *RDTBL* argument, i.e., use NIL.

6.6.1 Readtable Functions

(READTABLEP *RDTBL*) [Function]

Returns *RDTBL* if *RDTBL* is a real readtable (*not* T or ORIG), otherwise NIL.

(GETREADTABLE *RDTBL*) [Function]

If *RDTBL*=NIL, returns the primary read table. If *RDTBL*=T, returns the system terminal readtable. If *RDTBL* is a real readtable, returns *RDTBL*. Otherwise, generates an ILLEGAL READTABLE error.

(SETREADTABLE *RDTBL* *FLG*) [Function]

Sets the primary readtable to *RDTBL*. If *FLG*=T, SETREADTABLE sets the system terminal readtable, T. Note that the user can reset the other system readtables with SETQ, e.g., (SETQ FILERDTBL (GETREADTABLE)).

Generates an ILLEGAL READTABLE error if *RDTBL* is not NIL, T, or a real readtable. Returns the previous setting of the primary readtable, so SETREADTABLE is suitable for use with RESETFORM (page 9.20).

(COPYREADTABLE *RDTBL*) [Function]

Returns a copy of *RDTBL*. *RDTBL* can be a real readtable, NIL, T, or ORIG (in which case COPYREADTABLE returns a copy of the *original* system readtable), otherwise COPYREADTABLE generates an ILLEGAL READTABLE error.

Note that COPYREADTABLE is the only function that *creates* a readtable.

(RESETREADTABLE *RDTBL FROM*) [Function]

Copies (smashes) *FROM* into *RDTBL*. *FROM* and *RDTBL* can be NIL, T, or a real readtable. In addition, *FROM* can be ORIG, meaning use the system's original readtable.

6.6.2 Syntax Classes

A readtable is an object that contains information about the "syntax class" of each character. There are nine basic syntax classes: LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, ESCAPE, BREAKCHAR, SEPRCHAR, and OTHER, each associated with a primitive syntactic property. In addition, there is an unlimited assortment of user-defined syntax classes, known as "read-macros". The basic syntax classes are interpreted as follows:

LEFTPAREN (normally left parenthesis) Begins list structure.

RIGHTPAREN (normally right parenthesis) Ends list structure.

LEFTBRACKET (normally left bracket) Begins list structure. Also matches RIGHTBRACKET characters.

RIGHTBRACKET (normally left bracket) Ends list structure. Can close an arbitrary numbers of LEFTPAREN lists, back to the last LEFTBRACKET.

STRINGDELIM (normally double quote) Begins and ends text strings. Within the string, all characters except for the one(s) with class ESCAPE are treated as ordinary, i.e., interpreted as if they were of syntax class OTHER. To include the string delimiter inside a string, prefix it with the ESCAPE character.

ESCAPE (normally percent sign) Inhibits any special interpretation of the next character, i.e., the next character is interpreted to be of class OTHER, independent of its normal syntax class.

BREAKCHAR (None initially) Is a break character, i.e., delimits atoms, but is otherwise an ordinary character.

SEPRCHAR (space, carriage return, etc.) Delimits atoms, and is otherwise ignored.

OTHER Characters that are not otherwise special belong to the class OTHER.

Characters of syntax class LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, and STRINGDELIM are all *break* characters. That is, in addition to their interpretation as delimiting list or string structures, they also terminate the reading of an atom. Characters of class BREAKCHAR serve *only* to terminate atoms, with no other special meaning. In addition, if a break character is the first non-separator encountered by RATOM, it is read as a one-character atom. In order for a break character to be included in an atom, it must be preceded by the ESCAPE character.

Characters of class SEPRCHAR also terminate atoms, but are otherwise completely ignored; they can be thought of as logically spaces. As with break characters, they must be preceded by the ESCAPE character in order to appear in an atom.

For example, if $ were a break character and * a separator character, the input stream ABC**DEF$GH*$$ would be read by 6 calls to RATOM returning respectively ABC, DEF, $, GH, $, $.

Although normally there is only one character in a readtable having each of the list- and string-delimiting syntax classes (such as LEFTPAREN), it is perfectly acceptable for any character to have any syntax class, and for more than one to have the same class.

Note that a "syntax class" is an abstraction: there is no object referencing a collection of characters called a *syntax class*. Instead, a readtable provides the *association* between a character and its syntax class, and the input/output routines enforce the abstraction by using readtables to drive the parsing.

The functions below are used to obtain and set the syntax class of a character in a readtable. *CH* can either be a character code (a number), or a character (a single-character atom); those Interlisp objects that happen to be both, viz., one-digit numbers, are interpreted as character codes. For example, in Interlisp-10, 1 indicates control-A, and 49 indicates the *character* 1.

Note: Terminal tables, described in page 6.40, also associate characters with syntax classes, and they can also be manipulated with the functions below. The set of readtable and terminal table syntax classes are disjoint, so there is never any ambiguity about which type of table is being referred to.

(GETSYNTAX *CH TABLE*) [Function]

Returns the syntax class of *CH*, a character or a character code, with respect to *TABLE*. *TABLE* can be NIL, T, ORIG, or a real readtable or terminal table.

CH can also be a syntax class, in which case GETSYNTAX returns a list of the character codes in *TABLE* that have that syntax class.

(SETSYNTAX *CHAR CLASS TABLE*) [Function]

Sets the syntax class of *CHAR*, a character or character code, in *TABLE*. *TABLE* can be either NIL, T, or a real readtable or terminal table. SETSYNTAX returns the previous syntax class of *CHAR*. *CLASS* can be any one of the following:

- The name of one of the basic syntax classes.

- A list, which is interpreted as a read macro (see page 6.36).

- NIL, T, ORIG, or a real readtable or terminal table, which means to give *CHAR* the syntax class it has in the table indicated by *CLASS*. For example, (SETSYNTAX '%('ORIG *TABLE*) gives the left parenthesis character in *TABLE* the same syntax class that it has in the original system readtable.

- A character code or character, which means to give *CHAR* the same syntax class as the character *CHAR* in *TABLE*. For example, (SETSYNTAX '{ '%[*TABLE*) gives the left brace character the same syntax class as the left bracket.

(SYNTAXP *CODE CLASS TABLE*) [Function]

CODE is a character code; *TABLE* is NIL, T, or a real readtable or terminal table.

Returns T if *CODE* has the syntax class *CLASS* in *TABLE*; NIL otherwise.

CLASS can also be a read-macro type (MACRO, SPLICE, INFIX), or a read-macro option (FIRST, IMMEDIATE, etc.), in which case SYNTAXP returns T if the syntax class is a read-macro with the specified property.

Note: SYNTAXP will *not* accept a character as an argument, only a character *code.*

For convenience in use with SYNTAXP, the atom BREAK may be used to refer to *all* break characters, i.e., it is the union of LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, and BREAKCHAR. For purely symmetrical reasons, the atom SEPR corresponds to all separator characters. However, since the only separator characters are those that also appear in SEPRCHAR, SEPR and SEPRCHAR are equivalent.

Note that GETSYNTAX never returns BREAK or SEPR as a value although SETSYNTAX and SYNTAXP accept them as arguments. Instead, GETSYNTAX returns one of the disjoint basic syntax classes that comprise BREAK. BREAK as an argument to SETSYNTAX is interpreted to mean BREAKCHAR if the character is not already of one of the BREAK classes. Thus, if %(is of class LEFTPAREN, then (SETSYNTAX '%('BREAK) doesn't do anything, since %(is already a break character, but (SETSYNTAX '%('BREAKCHAR) means make %(be *just* a break character, and therefore disables the LEFTPAREN function of %(. Similarly, if one of the format characters is disabled completely, e.g., by (SETSYNTAX '%('OTHER), then (SETSYNTAX '%('BREAK) would make %(be *only* a break character; it would *not* restore %(as LEFTPAREN.

The following functions provide a way of collectively accessing and setting the separator and break characters in a readtable:

(GETSEPR *RDTBL*) [Function]

Returns a list of separator character codes in *RDTBL*. Equivalent to (GETSYNTAX 'SEPR *RDTBL*).

(GETBRK *RDTBL*) [Function]

Returns a list of break character codes in *RDTBL*. Equivalent to (GETSYNTAX 'BREAK *RDTBL*).

(SETSEPR *LST FLG RDTBL*) [Function]

Sets or removes the separator characters for *RDTBL*. *LST* is a list of charactors or character codes. *FLG* determines the action of SETSEPR as follows: If *FLG*=NIL, makes *RDTBL* have exactly the elements of *LST* as separators, discarding from *RDTBL* any old separator characters not in *LST*. If *FLG*=0, removes from *RDTBL* as separator characters all elements of *LST*. This provides an "UNSETSEPR". If *FLG*=1, makes each of the characters in *LST* be a separator in *RDTBL*.

If *LST*=T, the separator characters are reset to be those in the system's readtable for terminals, regardless of the value of *FLG*, i.e., (SETSEPR T) is equivalent to (SETSEPR (GETSEPR T)). If *RDTBL is* T, then the characters are reset to those in the original system table.

Returns NIL.

(SETBRK *LST FLG RDTBL*) [Function]

Sets the break characters for *RDTBL*. Similar to SETSEPR.

As with SETSYNTAX to the BREAK class, if any of the list- or string-delimiting break characters are disabled by an appropriate SETBRK (or by making it be a separator character), its special action for READ will *not* be restored by simply making it be a break character again with SETBRK. However, making these characters be break characters when they already *are* will have no effect.

The action of the ESCAPE character (normally %) is not affected by SETSEPR or SETBRK. It can be disabled by setting its syntax to the class OTHER, and other characters can be used for escape on input by assigning them the class ESCAPE. As of this writing, however, there is no way to change the output escape character; it is "hardwired" as %. That is, on output, characters of special syntax that need to be preceded by the ESCAPE character will always be preceded by %, independent of the syntax of % or which, if any characters, have syntax ESCAPE.

The following function can be used for defeating the action of the ESCAPE character or characters:

(ESCAPE *FLG RDTBL*) [Function]

If *FLG*=NIL, makes characters of class ESCAPE behave like characters of class OTHER on input. Normal setting is (ESCAPE T). ESCAPE returns the previous setting.

6.6.3 Read-Macros

Read-macros are user-defined syntax classes that can cause complex operations when certain characters are read. Read-macro characters are defined by specifying as a syntax class an expression of the form:

(*TYPE* *OPTION*$_1$ ··· *OPTION*$_N$ *FN*)

where *TYPE* is one of MACRO, SPLICE, or INFIX, and *FN* is the name of a function or a lambda expression. Whenever READ encounters a read-macro character, it calls the associated function, giving it as arguments the input file and readtable being used for that call to READ. The interpretation of the value returned depends on the type of read-macro:

MACRO

This is the simplest type of read macro. The result returned from the macro is treated as the expression to be read, instead of the read-macro character. Often the macro reads more input itself. For example, in order to cause ~EXPR to be read as (NOT EXPR), one could define ~ as

```
[MACRO (LAMBDA (FL RDTBL) (LIST 'NOT (READ FL RDTBL]
```

SPLICE

The result (which should be a list or NIL) is spliced into the input using NCONC. For example, if $ is defined by:

```
(SPLICE (LAMBDA NIL (APPEND FOO)))
```

and the value of FOO is (A B C), then when the user inputs (X $ Y), the result will be (X A B C Y).

INFIX

The associated function is called with a third argument, which is a list, in TCONC format (page 2.17), of what has been read at the current level of list nesting. The function's value is taken as a new TCONC list which replaces the old one. For example, + could be defined by:

```
(INFIX (LAMBDA (FL RDTBL Z)
               (RPLACA (CDR Z)
                       (LIST (QUOTE IPLUS)
                             (CADR Z)
                             (READ FL RDTBL)))
               Z))
```

If an INFIX read-macro character is encountered *not* in a list, the third argument to its associated function is NIL. If the function returns NIL, the read-macro character is essentially ignored and reading continues. Otherwise, if the function returns a TCONC list of one element, that element is the value of the READ. If it returns a TCONC list of more than one element, the list is the value of the READ.

The specification for a read-macro character can be augmented to specify various options *OPTION*$_1$ ··· *OPTION*$_N$, e.g., (MACRO FIRST IMMEDIATE *FN*). The following three disjoint options specify when the read-macro character is to be effective:

ALWAYS — The default. The read-macro character is always effective (except when preceded by the escape character), and is a break character, i.e., a member of (GETSYNTAX 'BREAK *RDTBL*).

FIRST — The character is interpreted as a read-macro character *only* when it is the first character seen after a break or separator character; in all other situations, the character is treated as having class OTHER. The read-macro character is *not* a break character. For example, the quote character is a FIRST read-macro character, so that DON'T is read as the single atom DON'T, rather than as DON followed by (QUOTE T).

ALONE — The read-macro character is *not* a break character, and is interpreted as a read-macro character only when the character would have been read as a separate atom if it were not a read-macro character, i.e., when its immediate neighbors are both break or separator characters. For example, * is an ALONE read-macro character in order to implement the comment pointer feature (see page 6.51).

Making a FIRST or ALONE read-macro character be a break character (with SETBRK) disables the read-macro interpretation, i.e., converts it to syntax class BREAKCHAR. Making an ALWAYS read-macro character be a break character is a no-op.

The following two disjoint options control whether the read-macro character is to be protected by the ESCAPE character on output:

ESCQUOTE or ESC — The default. When printed with PRIN2, the read-macro character will be preceded by the output escape character (%).

NOESCQUOTE or NOESC — The read-macro character will be printed without an escape, e.g., ' is a NOESCQUOTE character. Unless you are very careful what you are doing, read-macro characters in FILERDTBL should never be NOESCQUOTE, since symbols that happen to contain the read-macro character will not read back in correctly.

The following two disjoint options control when the macro's function is actually executed:

IMMEDIATE or IMMED

The read-macro character is immediately activated, i.e., the current line is terminated, as if an EOL had been typed, a carriage-return line-feed is printed, and the entire line (including the macro character) is passed to the input function.

IMMEDIATE read-macro characters enable the user to specify a character that will take effect immediately, as soon as it is encountered in the input, rather than waiting for the line to be terminated. Note that this is not necessarily as soon as the character is *typed*. Characters that cause action as soon as they are typed are interrupt characters (see page 9.17).

Note that since an IMMEDIATE macro causes any input before it to be sent to the reader, characters typed before an IMMEDIATE read-macro character cannot be erased by control-A or control-Q once the IMMEDIATE character has been typed, since they have already passed through the line buffer. However, an INFIX read macro can still alter some of what has been typed earlier, via its third argument.

NONIMMEDIATE or NONIMMED

The default. The read-macro character is a normal character with respect to the line buffering, and so will not be activated until a carriage-return or matching right parenthesis or bracket is seen.

Making a read-macro character be both ALONE and IMMEDIATE is a contradiction, since ALONE requires that the next character be input in order to see if it is a break or separator character. Thus, ALONE read-macros are always NONIMMEDIATE, regardless of whether or not IMMEDIATE is specified.

Read-macro characters can be "nested". For example, if = is defined by

```
(MACRO (LAMBDA (FL RDTBL) (EVAL (READ FL RDTBL))))
```

and ! is defined by

```
(SPLICE (LAMBDA (FL RDTBL) (READ FL RDTBL)))
```

then if the value of FOO is (A B C), and (X =FOO Y) is input, (X (A B C) Y) will be returned. If (X !=FOO Y) is input, (X A B C Y) will be returned.

If a read-macro's function calls READ, and the READ returns NIL, the function cannot distinguish the case where a RIGHTPAREN or RIGHTBRACKET followed the read-macro character, (e.g. "(A B ')"), from the case where the atom NIL (or "()") actually appeared. Therefore, in Interlisp-10, reading a single RIGHTPAREN or RIGHTBRACKET via a READ inside of a read-macro function is disallowed. If this occurs, the paren/bracket is put back into the input buffer, and a READ-MACRO CONTEXT ERROR is generated. The following two functions are useful for avoiding this error:

(INREADMACROP) [Function]

Returns NIL if currently *not* under a read-macro function, otherwise the number of unmatched left parentheses or brackets.

(SETREADMACROFLG *FLG*) [Function]

Resets the "read-macro" flag, i.e., the internal system flag that informs READ that it is under a read macro function, and causes it to generate a READ-MACRO CONTEXT ERROR, if an unmatched) or] is encountered. Returns the previous value of the flag. The main use for this is when debugging read-macro functions: to avoid spurious READ-MACRO CONTEXT error messages when typing into breaks, the user can put (SETREADMACROFLG) on BREAKRESETFORMS (page 9.13).

The READ-MACRO CONTEXT error does not occur in Interlisp-D; a READ inside of a read-macro when the next input character is a RIGHTPAREN or RIGHTBRACKET eats the character and returns NIL, just as if the READ had not occurred inside a read-macro.

If a call to READ from within a read-macro encounters an unmatched RIGHTBRACKET *within* a list, the bracket is simply put back into the buffer to be read (again) at the higher level. Thus, inputting an expression such as (A B '(C D] works correctly.

(READMACROS *FLG RDTBL*) [Function]

If *FLG*=NIL, turns off action of read-macros. If *FLG*=T, turns them on. Returns previous setting.

In Interlisp-D, turns off/on action of read-macros in readtable *RDTBL*.

The following read macros are standardly defined in Interlisp:

' (single-quote)

Currently defined only in T and EDITRDTBL. Returns the next expression, wrapped in a call to QUOTE; e.g., 'FOO reads as (QUOTE FOO). The macro is defined as a FIRST read macro, so that the quote character has no effect in the middle of a symbol. The macro is also ignored if the quote character is immediately followed by a separator character.

control-Y

Defined in T and EDITRDTBL. Returns the result of evaluating the next expression. For example, if the value of FOO is (A B), then (LIST 1 *control-Y*FOO 2) is read as (1 (A B) 2), but note that no structure is copied; the CADR of that input expression is still EQ to the value of FOO. Control-Y can thus be used to read structures that ordinarily have no read syntax. For example, the value returned from reading (KEY1 *control-Y*(ARRAY 10)) has an array as its second element. Control-Y can be thought of as an "un-quote" character. The choice of character to perform this function is changeable with SETTERMCHARS (page 17.59).

‘ (back-quote)

Back-quote makes it easier to write programs to construct complex data structures. Back-quote is like quote, except that within the back-quoted expression, forms can be evaluated. The general idea is that the back-quoted expression is a "template" containing some constant parts (as with a quoted form) and some parts to be filled in by evaluating something.

Within the back-quoted expression, the character "," (comma) introduces a form to be evaluated. A form preceded by ",@" is to be spliced in, using APPEND, and a form preceded by ",." is to be spliced in, using NCONC. Unlike with control-Y, however, the evaluation occurs not at the time the form is read, but at the time the back-quoted expression is evaluated. That is, the back-quote macro returns an expression which, when evaluated, produces the desired structure.

For example, if the value of FOO is (1 2 3 4), then the form ‘(A ,(CAR FOO) ,@(CDDR FOO) D E) evaluates to (A 1 3 4 D E); it is logically equivalent to writing (CONS 'A (CONS (CAR FOO) (APPEND (CDDR FOO) '(D E)))). Back-quote is particularly useful for writing compiler macros. For example,

```
`(COND
   ((FIXP ,(CAR X))
      ,(CADR X))
   (T ,@(CDDR X)))
```

is equivalent to writing

```
(LIST 'COND
     (LIST (LIST 'FIXP (CAR X))
           (CADR X))
     (CONS 'T (CDDR X)))
```

Note that comma does *not* have any special meaning outside of a back-quote context.

For users without a back-quote character on their keyboards, back-quote can also be written as |' (vertical-bar, quote). In Interlisp-D, back-quote is typed as shift-linefeed.

? Defined in T and EDITRDTBL. Implements the ?= command for on-line help regarding the function currently being "called" in the typein (see page 9.5).

* Defined in FILERDTBL only. Implements the comment pointer feature for saving space by keeping the text of comments outside memory (page 6.51).

control-W Defined in T and EDITRDTBL, Interlisp-10 only. An IMMEDIATE read macro that deletes the previous expression. In Interlisp-D, control-W is an editing character that deletes the previous "word".

| (vertical bar) When followed by ' (quote), is a synonym for back-quote; followed by certain other characters, it is used by HPRINT and HREAD to print and read in unusual expressions; otherwise is ignored, i.e., treated as a separator character, enabling the editor's CHANGECHAR feature (page 6.55).

6.7 TERMINAL TABLES

A readtable contains input/output information that is *media-independent.* For example, the action of parentheses is the same regardless of the device from which the input is being performed. A terminal table is an object that contains information that pertains to *terminal* input/output operations only, such as the character to type to delete the last character or to delete the last line. In addition, terminal tables contain such information as how line-buffering is to be performed, how control characters are to be echoed/printed, whether lower case input is to be converted to upper case, etc.

Using the functions below, the user may change, reset, or copy terminal tables, or create a new terminal table and install it as the primary terminal table via SETTERMTABLE. However, unlike readtables, terminal tables cannot be passed as arguments to input/output functions.

6.7.1 Terminal Table Functions

(TERMTABLEP *TTBL*) [Function]

Returns *TTBL*, if *TTBL* is a real terminal table, NIL otherwise.

(GETTERMTABLE *TTBL*) [Function]

If *TTBL*=NIL, returns the primary (i.e., current) terminal table. If *TTBL* is a real terminal table, return *TTBL*. Otherwise, generates an ILLEGAL TERMINAL TABLE error.

(SETTERMTABLE *TTBL*) [Function]

Sets the primary terminal table to be *TTBL*. Returns the previous *TTBL*. Generates an ILLEGAL TERMINAL TABLE error if *TTBL* is not a real terminal table.

(COPYTERMTABLE *TTBL*) [Function]

Returns a copy of *TTBL*. *TTBL* can be a real terminal table, NIL, or ORIG, in which case it returns a copy of the *original* system terminal table. Note that COPYTERMTABLE is the only function that *creates* a terminal table.

(RESETTERMTABLE *TTBL FROM*) [Function]

Copies (smashes) *FROM* into *TTBL*. *FROM* and *TTBL* can be NIL or a real terminal table. In addition, *FROM* can be ORIG, meaning to use the system's original terminal table.

6.7.2 Terminal Syntax Classes

A terminal table associates with each character a single "terminal syntax class", one of CHARDELETE, LINEDELETE, WORDDELETE (Interlisp-D only), RETYPE, CTRLV, EOL, and NONE. Unlike readtable classes, only one character in a particular terminal table can belong to each of the classes (except for the default class NONE). When a new character is assigned one of these syntax classes by SETSYNTAX, the previous character is disabled (i.e., reassigned the syntax class NONE), and the value of SETSYNTAX is the code for the previous character of that class, if any, otherwise NIL.

The terminal syntax classes are interpreted as follows:

CHARDELETE or DELETECHAR

(Initially control-A under Tenex, del under Tops20, BackSpace in Interlisp-D) Typing this character deletes the previous character typed. Repeated use of this character deletes successive characters back to the beginning of the line.

LINEDELETE or DELETELINE

(Initially control-Q in Interlisp-10 under Tenex and in Interlisp-D, control-U under Tops20) Typing this character deletes the whole line; it cannot be used repeatedly.

WORDDELETE

(Interlisp-D only; initially control-W) Typing this character deletes the previous "word", i.e., sequence of non-separator characters.

RETYPE

(Initially control-R) Causes the line to be retyped as Interlisp sees it (useful when repeated deletions make it difficult to see what remains).

CTRLV or CNTRLV (Initially control-V) When followed by A, B, ··· Z, inputs the corresponding control character control-A, control-B, ··· control-Z. This allows interrupt characters to be input without causing an interrupt.

EOL On input from a terminal, the EOL character signals to the line buffering routine to pass the input back to the calling function. It also is used to terminate inputs to READLINE (page 8.30). In general, whenever the phrase carriage-return linefeed is used, what is meant is the character with terminal syntax class EOL.

NONE The terminal syntax class of all other characters.

GETSYNTAX, SETSYNTAX, and SYNTAXP all work on terminal tables as well as readtables (see page 6.34). When given NIL as a *TABLE* argument, GETSYNTAX and SYNTAXP use the primary readtable or primary terminal table depending on which table contains the indicated *CLASS* argument. For example, (SETSYNTAX *CH* 'BREAK) refers to the primary readtable, and (SETSYNTAX *CH* 'CHARDELETE) refers to the primary terminal table. In the absence of such information, all three functions default to the primary readtable; e.g., (SETSYNTAX '{ '%[) refers to the primary read table. If given incompatible *CLASS* and table arguments, all three functions generate errors. For example, (SETSYNTAX *CH* 'BREAK *TTBL*), where *TTBL* is a terminal table, generates an ILLEGAL READTABLE error, and (GETSYNTAX 'CHARDELETE *RDTBL*) generates an ILLEGAL TERMINAL TABLE error.

6.7.3 Terminal Control Functions

(ECHOCONTROL *CHAR MODE TTBL*) [Function]

Used to indicate how control characters are to be echoed or printed. *CHAR* is a character or character code. *MODE* may be one of the atoms IGNORE, REAL, SIMULATE, or INDICATE,[9] which specify how the control character should be printed:

IGNORE *CHAR* is never printed.

REAL *CHAR* itself is printed; i.e., the raw control character is sent to the terminal. Some terminals, particularly displays, respond to certain control characters in interesting ways.

SIMULATE Output of *CHAR* is simulated. For example, control-I (tab) may be simulated by printing spaces. The simulation is machine-specific and beyond the control of the user.

INDICATE *CHAR* is printed as ↑ followed by the corresponding alphabetic character.

The value of ECHOCONTROL is the previous output mode for *CHAR*. If *MODE*=NIL, ECHOCONTROL returns the current output mode without changing it.

Note that although the name of this function suggests echoing only, it affects *all* output of the control character, both echoing of input and printing of output.

[9] UPARROW is an obsolete synonym of INDICATE.

The two cannot be specified independently, which can lead to some trickiness in DELETECONTROL messages (below).

In Interlisp-10, echoing information can be specified only for control characters (although *all* echoing can be disabled using ECHOMODE, below). Therefore, if *CHAR* is an alphabetic character (or code), it refers to the corresponding control character, e.g., (ECHOCONTROL 'Z 'INDICATE) makes control-Z echo as ↑Z. All other values of *CHAR* generate ILLEGAL ARG errors. In Interlisp-D and Interlisp-VAX, it is possible to specify echoing information for *all* characters, using the function ECHOCHAR.

(ECHOCHAR *CHARCODE MODE TTBL*) [Function]

(Interlisp-D, Interlisp-VAX only) Like ECHOCONTROL, but *CHARCODE* must be a character code, and can specify *any* character—no coercions are performed. The INDICATE mode for "meta" characters, i.e., characters whose codes are in the range 200Q through 377Q, causes the character to be printed following a #. For example, meta-A would print as #A, meta-control-B as #↑B.

CHARCODE can also be a list of characters, in which case ECHOCHAR is applied to each of them with arguments *MODE* and *TTBL*.

(ECHOMODE *FLG TTBL*) [Function]

If *FLG*=T, turns echoing for terminal table *TTBL* on. If *FLG*=NIL, turns echoing off. Returns the previous setting.

(GETECHOMODE *TTBL*) [Function]

Returns the current echo mode for *TTBL*.

(DELETECONTROL *TYPE MESSAGE TTBL*) [Function]

Specifies the output protocol when a CHARDELETE or LINEDELETE is typed. In the case of character deletion, Interlisp-10 is initially set up for hardcopy terminals: it echos the characters being deleted, preceding the first by a \ and following the last by a \, so that it is easy to see exactly what was deleted, viz., the characters between the \'s. Interlisp-D and Interlisp-VAX are initially set up to physically erase the deleted characters from the display, backing up over them. The various values of *TYPE* specify different phases of the deletion, as follows:

1STCHDEL *MESSAGE* is the message printed the first time CHARDELETE is typed. Initially "\" in Interlisp-10.

NTHCHDEL *MESSAGE* is the message printed on subsequent CHARDELETE's (without intervening characters). Initially "" in Interlisp-10.

POSTCHDEL *MESSAGE* is the message printed when input is resumed following a sequence of one or more CHARDELETE's. Initially "\" in Interlisp-10.

EMPTYCHDEL *MESSAGE* is the message printed when a CHARDELETE is typed and there are no characters in the buffer. Initially "##^{cr}" in Interlisp-10.

ECHO The characters deleted by CHARDELETE are echoed. *MESSAGE* is ignored.

NOECHO — The characters deleted by CHARDELETE are not echoed *MESSAGE* is ignored.

LINEDELETE — *MESSAGE* is the message printed when LINEDELETE character is typed. Initially "##^{cr}".

Note: In Interlisp-10, the LINEDELETE, 1STCHDEL, NTHCHDEL, POSTCHDEL, and EMPTYCHDEL messages must be 4 characters or fewer in length.

DELETECONTROL returns the previous message as a string. If *MESSAGE*=NIL, the value returned is the previous message without changing it. For ECHO and NOECHO, the value of DELETECONTROL is the previous echo mode, i.e., ECHO or NOECHO.

(GETDELETECONTROL *TYPE* *TTBL*) [Function]

Returns the current DELETECONTROL mode for *TYPE* in *TTBL*.

If the user's terminal is a display, DELETECONTROL and ECHOCONTROL can be used to make it really delete the last character by performing the following:

```
(ECHOCONTROL 8 'REAL)
```

8 is the code for control-H, which is backspace; we want the terminal to really backspace when we send ↑H.

```
(DELETECONTROL 'NOECHO)
```

Do not echo the deleted characters.

```
(DELETECONTROL '1STCHDEL "↑H ↑H")
(DELETECONTROL 'NTHCHDEL "↑H ↑H")
```

Erase each character by backspacing over it, printing a space, then backspacing again to put the carriage in the right place.

The following functions manipulate the RAISE mode, which determines whether lower case characters are converted to upper case when input from the terminal. (There currently is no "raise" mode for input from files.)

(RAISE *FLG* *TTBL*) [Function]

Sets the RAISE mode for terminal table *TTBL*. If *FLG*=NIL, all characters are passed as typed. If *FLG*=T, input is echoed as typed, but lowercase letters are converted to upper case. If *FLG*=0, input is converted to upper case *before* it is echoed. Returns the previous setting.[10]

[10] In Interlisp-10, both (RAISE) and (RAISE T) execute Tenex/Tops20 JSYS calls corresponding to the Executive command NORAISE, while (RAISE 0) executes the JSYS calls corresponding to the Executive command RAISE. Thus with (RAISE T), the conversion to uppercase is performed by Interlisp, while with (RAISE 0) the conversion is performed at the operating system level, i.e., before Interlisp-10 even sees the characters. The initial setting of RAISE in Interlisp-10 is determined by the terminal mode at the time the user first starts up the system. When a SYSOUT is started, the RAISE mode is restored to whatever it was prior to the SYSOUT.

(GETRAISE *TTBL*) [Function]

Returns the current RAISE mode for *TTBL*.

6.7.4 Line-Buffering

Characters typed at the terminal are stored in two buffers before they are passed to an input function. All characters typed in are put into the low-level "system buffer", which allows type-ahead. When an input function is entered, characters are transferred to the "line buffer" until a character with terminal syntax class EOL appears (or, for calls from READ, when the count of unbalanced open parentheses reaches 0).[11] Until this time, the user can delete characters one at a time from the line buffer by typing the current CHARDELETE character, or delete the entire line buffer back to the last carriage-return by typing the current LINEDELETE.

Note that this line editing is *not* performed by READ or RATOM, but by Interlisp, i.e., it does not matter (nor is it necessarily known) which function will ultimately process the characters, only that they are still in the Interlisp line buffer. However, the function that is requesting input at the time the buffering starts does determine whether parentheses counting is observed. For example, if a program performs (PROGN (RATOM) (READ)) and the user types in "A (B C D)", the user must type in the carriage-return following the right parenthesis before any action is taken, because the line buffering is happening under RATOM. If the program had performed (PROGN (READ) (READ)), the line-buffering would be under READ, so that the right parenthesis would terminate line buffering, and no terminating carriage-return would be required.

Once a carriage-return has been typed, the entire line is "available" even if not all of it is processed by the function initiating the request for input. If any characters are "left over", they are returned immediately on the next request for input. For example, (LIST (RATOM) (READC) (RATOM)) when the input is "A B^{cr}" returns the three-element list (A % B) and leaves the carriage-return in the buffer.

If a carriage-return is typed when the input under READ is not "complete" (the parentheses are not balanced or a string is in progress), line buffering continues, but the lines completed so far are not available for editing with CHARDELETE or LINEDELETE.

The function CONTROL is available to defeat line-buffering:

(CONTROL *MODE TTBL*) [Function]

If *MODE*=T, eliminates Interlisp's normal line-buffering for the terminal table *TTBL*. If *MODE*=NIL, restores line-buffering (normal). When operating with a terminal table in which (CONTROL T) has been performed, characters are returned to the calling function without line-buffering as described below.

CONTROL returns its previous setting.

(GETCONTROL *TTBL*) [Function]

Returns the current control mode for *TTBL*.

The function that initiates the request for input determines how the line is treated when (CONTROL T) is in effect:

[11] PEEKC is an exception; it returns the character immediately when its second argument is NIL.

READ

If the expression being typed is a list, the effect is the same as though done with (CONTROL NIL);-i.e., line-buffering continues until a carriage-return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered, e.g., (READ) when the input is "ABC<space>" immediately returns ABC. CHARDELETE and LINEDELETE are available on those characters still in the buffer. Thus, if a program is performing several reads under (CONTROL T), and the user types "NOW IS THE TIME" followed by control-Q, only TIME is deleted, since the rest of the line has already been transmitted to READ and processed.

An exception to the above occurs when the break or separator character is an opening parenthesis, bracket or double-quote, since returning at this point would leave the line buffer in a "funny" state. Thus if the input to (READ) is "ABC(", the ABC is not read until a carriage-return or matching parentheses is encountered. In this case the user could LINEDELETE the entire line, since all of the characters are still in the buffer.

RATOM

Characters are returned as soon as a break or separator character is encountered. Until then, LINEDELETE and CHARDELETE may be used as with READ. For example, (RATOM) followed by "ABC<control-A><space>" returns AB. (RATOM) followed by "(<control-A>" returns (and types ## indicating that control-A was attempted with nothing in the buffer, since the (is a break character and would therefore already have been read.

READC or PEEKC

The character is returned immediately; no line editing is possible. In particular, (READC) is perfectly happy to return the CHARDELETE or LINEDELETE characters, or the ESCAPE character (%).

The system buffer and line buffer can be directly manipulated using the following functions.

(CLEARBUF *FILE FLG*) [Function]

Clears the input buffer for *FILE*. If *FILE* is T and *FLG* is T, the contents of Interlisp's system buffer and line buffer are saved (and can be obtained via SYSBUF and LINBUF described below).

When control-D or control-E is typed, or any of the interrupt characters that require terminal interaction is typed (control-H, control-P, or control-S), Interlisp automatically performs (CLEARBUF T T). For control-P, control-S, and, when the break is exited normally, control-H, Interlisp restores the buffer after the interaction.

The action of (CLEARBUF T), i.e., clearing of typeahead, is also available as the RUBOUT interrupt character, initially assigned to the del key in Interlisp-D and in Interlisp-10 under Tenex, control-Z under Tops20. Note that this interrupt clears both buffers at the time it is *typed*, whereas the action of the CHARDELETE and LINEDELETE character occur at the time they are *read*.

(SYSBUF *FLG*) [Function]

If *FLG*=T, returns the contents of the system buffer (as a string) that was saved at the last (CLEARBUF T T). If *FLG*=NIL, clears this internal buffer.

(LINBUF *FLG*) [Function]

Same as SYSBUF for the line buffer.

If both the system buffer and Interlisp's line buffer are empty, the internal buffers associated with LINBUF and SYSBUF are not changed by a (CLEARBUF T T).

(BKSYSBUF *X FLG RDTBL*) [Function]

BKSYSBUF sets the system buffer to the PRIN1-name of *X*. The effect is the same as though the user typed *X*. Some implementations have a limit on the length of *X*, in which case characters in *X* beyond the limit are ignored. Returns *X*.

If *FLG* is *T*, then the PRIN2-name of *X* is used, computed with respect to the readtable *RDTBL*.

Note that if the user is typing at the same time as the BKSYSBUF is being performed, the relative order of the type-in and the characters of *X* is unpredictable.

Compatibility note: Some implementations of BKSYSBUF (Interlisp-10) use a "system" buffer, from which keyboard interrupts are also processed. In this case, BKSYSBUF of an interrupt character actually invokes the interrupt at some (asynchronous) time after the BKSYSBUF is initiated. In other implementations (Interlisp-D), the characters are not processed for interrupts, and it is possible to BKSYSBUF characters which would otherwise be impossible to type.

(BKLINBUF *STR*) [Function]

STR is a string. BKLINBUF sets Interlisp's line buffer to *STR*. Some implementations have a limit on the length of *STR*, in which case characters in *STR* beyond the limit are ignored. Returns *STR*.

BKLINBUF, BKSYSBUF, LINBUF, and SYSBUF provide a way of "undoing" a CLEARBUF. Thus to "peek" at various characters in the buffer, one could perform (CLEARBUF T T), examine the buffers via LINBUF and SYSBUF, and then put them back.

The more common use of these functions is in saving and restoring typeahead when a program requires some unanticipated (from the user's standpoint) input. The function RESETBUFS provides a convenient way of simply clearing the input buffer, performing an interaction with the user, and then restoring the input buffer.

(RESETBUFS $FORM_1$ $FORM_2$ ··· $FORM_N$) [NLambda NoSpread Function]

Clears any typeahead (ringing the terminal's bell if there was, indeed, typeahead), evaluates $FORM_1$, $FORM_2$,··· $FORM_N$, then restores the typeahead. Returns the value of $FORM_N$. Compiles open.

6.8 PRETTYPRINT

The standard way of printing out function definitions (on the terminal or into files) is to use PRETTYPRINT.

(PRETTYPRINT *FNS PRETTYDEFLG —*) [Function]

FNS is a list of functions. If *FNS* is atomic, its value is used). The definitions of the functions are printed in a pretty format on the primary output file using the primary readtable. For example, if FACTORIAL were defined by typing

```
(DEFINEQ (FACTORIAL [LAMBDA (N) (COND ((ZEROP N) 1)
(T (ITIMES N (FACTORIAL (SUB1 N]
```

(PRETTYPRINT '(FACTORIAL)) would print out

```
(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES N (FACTORIAL (SUB1 N])
```

PRETTYDEFLG is T when called from PRETTYDEF (and hence MAKEFILE). Among other actions taken when this argument is true, PRETTYPRINT indicates its progress in writing the current output file: whenever it starts a new function, it prints on the terminal the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

PRETTYPRINT operates correctly on functions that are BROKEN, BROKEN-IN, ADVISED, or have been compiled with their definitions saved on their property lists: it prints the original, pristine definition, but does not change the current state of the function. If a function is not defined but is known to be on one of the files noticed by the file package, PRETTYPRINT loads in the definition (using LOADFNS) and prints it (except when called from PRETTYDEF). If PRETTYPRINT is given an atom which is not the name of a function, but has a value, it prettyprints the value. Otherwise, PRETTYPRINT attempts spelling correction. If all fails, PRETTYPRINT returns (*FN* NOT PRINTABLE).

(PP *FN*$_1$ ··· *FN*$_N$) [NLambda NoSpread Function]

For prettyprinting functions to the terminal. PP calls PRETTYPRINT with the primary output file set to T and the primary read table set to T. The primary output file and primary readtable are restored after printing.

(PP FOO) is equivalent to (PRETTYPRINT '(FOO)); (PP FOO FIE) is equivalent to (PRETTYPRINT '(FOO FIE)).

As described above, when PRETTYPRINT, and hence PP, is called with the name of a function that is not defined, but whose definition is on a file known to the file package, the definition is automatically read in and then prettyprinted. However, if the user does not intend on editing or running the definition, but simply wants to *see* the definition, the function PF described below can be used to simply copy the corresponding characters from the file to the terminal. This results in a savings in both space and time, since it is not necessary to allocate storage to actually read in the definition, and it is not necessary to re-prettyprint it (since the function is already in prettyprint format on the file).

(PF *FN* *FROMFILES* *TOFILE*) [NLambda NoSpread Function]

Copies the definition of *FN* found on each of the files in *FROMFILES* to *TOFILE*. If *TOFILE*=NIL, defaults to T. If *FROMFILES*=NIL, defaults to (WHEREIS *FN* NIL T) (see page 11.10). The typical usage of PF is simply to type "PF *FN*".

When printing to the terminal, PF performs several transformations on the characters in the file that comprise the definition for *FN*: (1) font information (page 6.55) is stripped out (except in Interlisp-D, whose display supports multiple fonts); (2) occurrences of the CHANGECHAR (page 6.55) are not printed; (3) since functions typically tend to be printed to a file with a larger linelength than when printing to a terminal, the number of leading spaces on each line is cut in half;[12] and (4) comments are elided, if **COMMENT**FLG is non-NIL (see page 6.50).

While the function PRETTYPRINT prints entire function definitions, the function PRINTDEF can be used to print parts of functions, or arbitrary Interlisp structures:

(PRINTDEF *EXPR LEFT DEF TAILFLG FNSLST FILE*) [Function]

Prints the expression *EXPR* in a pretty format on *FILE* using the primary readtable. *LEFT* is the left hand margin (LINELENGTH determines the right hand margin.)[13]

DEF=T means *EXPR* is a function definition, or a piece of one. If *DEF*=NIL, no special action is taken for LAMBDA's, PROG's, COND's, comments, CLISP, etc. *DEF* is NIL when PRETTYDEF calls PRETTYPRINT to print variables and property lists, and when PRINTDEF is called from the editor via the command PPV.

TAILFLG=T means *EXPR* is interpreted as a tail of a list, to be printed without parentheses.

FNSLST is for use with the Font package (page 6.55). PRINTDEF prints occurrences of any function in the list *FNSLST* in a different font, for emphasis. MAKEFILE passes as *FNSLST* the list of all functions on the file being made.

6.8.1 Comment Feature

A facility for annotating Interlisp functions is provided in PRETTYPRINT. Any expression beginning with the atom * is interpreted as a comment and printed in the right margin. Example:

```
(FACTORIAL
  [LAMBDA (N)                               (* COMPUTES N!)
     (COND
       ((ZEROP N)                           (* 0!=1)
         1)
       (T                                   (* RECURSIVE DEFINITION:
                                               N!=N*N-1!)
        (ITIMES N (FACTORIAL (SUB1 N])
```

These comments actually form a part of the function definition. Accordingly, * is defined as an nlambda nospread function that returns its argument, similar to QUOTE. When running an interpreted function, * is entered the same as any other Interlisp function. Therefore, comments should only be placed where they will not harm the computation, i.e., where a quoted expression could be placed. For example, writing

```
(ITIMES N (FACTORIAL (SUB1 N)) (* RECURSIVE DEFINITION))
```

[12] Unless PFDEFAULT is T. PFDEFAULT is initially NIL.

[13] PRINTDEF initially performs (TAB *LEFT* T), which means to space to position *LEFT*, unless already beyond this position, in which case it does nothing.

in the above function would cause an error when ITIMES attempted to multiply N, N-1!, and RECURSIVE.

For compilation purposes, * is defined as a macro which compiles into no instructions (unless the comment has been placed where it has been used for value, in which case the compiler prints an appropriate error message and compiles * as QUOTE). Thus, the compiled form of a function with comments does not use the extra atom and list structure storage required by the comments in the source (interpreted) code. This is the way the comment feature is intended to be used.

A comment of the form (* E *x*) causes *x* to be evaluated at prettyprint time, as well as printed as a comment in the usual way. For example, (* E (RADIX 8)) as a comment in a function containing octal numbers can be used to change the radix to produce more readable printout.

The comment character * is stored in the variable COMMENTFLG. The user can set it to some other value, e.g. ";", and use this to indicate comments.

COMMENTFLG [Variable]

If CAR of an expression is EQ to COMMENTFLG, the expression is treated as a comment by PRETTYPRINT. COMMENTFLG is initialized to *. Note that whatever atom is chosen for COMMENTFLG should also have an appropriate function definition and compiler macro, for example, by copying those of *.

Comments are designed mainly for documenting *listings*. Therefore, when prettyprinting to the terminal, comments are suppressed and printed as the string **COMMENT**. The value of **COMMENT**FLG determines the action.

COMMENTFLG [Variable]

If **COMMENT**FLG is NIL, comments are printed. Otherwise, the value of **COMMENT**FLG is printed. Initially " **COMMENT** ".

The functions PP* and PF* are provided to easily print functions, including their comments.

(PP* *x*) [NLambda NoSpread Function]

PP* operates exactly like PP except it first sets **COMMENT**FLG to NIL, so comments are printed in full.

(PF* *FN FROMFILES TOFILE*) [NLambda NoSpread Function]

PF* operates exactly like PF except it first sets **COMMENT**FLG to NIL, so comments are printed in full.

(COMMENT1 *L* —) [Function]

Prints the comment *L*. COMMENT1 is a separate function[14] to permit the user to write prettyprint macros (page 6.54) that use the regular comment printer. For example, to cause comments to be printed at a larger than normal linelength, one could put an entry for * on PRETTYPRINTMACROS:

```
(* LAMBDA (X) (RESETFORM (LINELENGTH 100) (COMMENT1 X)))
```

[14] COMMENT1 is an entry to the PRETTYPRINT block. However, it is called internally by PRETTYPRINT so that advising or redefining it will not affect the action of PRETTYPRINT. COMMENT1 should *not* be called when not under a PRINTDEF.

This macro resets the line length, prints the comment, and then restores the line length.

COMMENT1 expects to be called from within the environment established by PRINTDEF, so ordinarily the user should call it *only* from within prettyprint macros.

6.8.2 Comment Pointers

For a well-commented collection of programs, the list structure, atom, and pname storage required to represent the comments in core can be significant. If the comments already appear on a file and are not needed for editing, a significant savings in storage can be achieved by simply leaving the text of the comment on the file when the file is loaded, and instead retaining in core only a *pointer* to the comment. This feature has been implemented by defining * as a read-macro in FILERDTBL which, instead of reading in the entire text of the comment, constructs an expression containing (1) the name of the file in which the text of the comment is contained, (2) the address of the first character of the comment, (3) the number of characters in the comment, and (4) a flag indicating whether the comment appeared at the right hand margin or centered on the page. For output purposes, * is defined on PRETTYPRINTMACROS (page 6.54) so that it prints the comments represented by such pointers by simply copying the corresponding characters from one file to another, or to the terminal. Normal comments are processed the same as before, and can be intermixed freely with comment pointers.

The comment pointer feature is controlled by the value of NORMALCOMMENTSFLG.

NORMALCOMMENTSFLG [Variable]

The comment pointer feature is enabled by setting NORMALCOMMENTSFLG to NIL. NORMALCOMMENTSFLG is initially T.

NORMALCOMMENTSFLG can be changed as often as desired. Thus, some files can be loaded normally, and others with their comments converted to comment pointers.

For convenience of editing selected comments, an edit macro, GET*, is included, which loads in the text of the corresponding comment. The editor's PP* command, in contrast, prints the comment *without* reading it by simply copying the corresponding characters to the terminal. GET* is defined in terms of GETCOMMENT:

(GETCOMMENT *X DESTFL* —) [Function]

If *X* is a comment pointer, replaces *X* with the actual text of the comment, which it reads from its file. Returns *X* in all cases. If *DESTFL* is non-NIL, it is the name of an open file, to which GETCOMMENT copies the comment; in this case, *X* remains a comment pointer, but it has been changed to point to the new file (unless NORMALCOMMENTSFLG = DONTUPDATE).

(PRINTCOMMENT *X*) [Function]

Defined as the prettyprint macro for *: copies the comment to the primary output file by using GETCOMMENT.

(READCOMMENT *FL RDTBL LST*) [Function]

Defined as the read macro for * in FILERDTBL: if NORMALCOMMENTSFLG is NIL,

- it constructs a comment pointer.[15]

Note that a certain amount of care is required in using the comment pointer feature. Since the text of the comment resides on the file pointed to by the comment pointer, that file must remain in existence as long as the comment is needed. GETCOMMENT helps out by changing the comment pointer to always point at the most recent file that the comment lives on. However, if the user has been performing repeated MAKEFILE's (page 11.6) in which differing functions have changed at each invocation of MAKEFILE, it is possible for the comment pointers in memory to be pointing at several versions of the same file, since a comment pointer is only updated when the function it lives in is prettyprinted, not when the function has been copied verbatim to the new file. This can be a problem for file systems, such as Tenex and Tops20, that have a built-in limit on the number of versions of a given file that will be made before old versions are expunged. In such a case, the user should set the version retention count of any directories involved to be infinite. GETCOMMENT prints an error message if the file that the comment pointer points at has disappeared.

Similarly, one should be cognizant of comment pointers in SYSOUTs, and be sure to retain any files thus pointed to.

When using comment pointers, the user should also not set PRETTYFLG (page 6.54) to NIL or call MAKEFILE with option FAST, since this will prevent functions from being prettyprinted, and hence not get the text of the comment copied into the new file.

If the user changes the value of COMMENTFLG but still wishes to use the comment pointer feature, the new COMMENTFLG should be given the same read-macro definition in FILERDTBL as * has, and the same entry be put on PRETTYPRINTMACROS. For example, if COMMENTFLG is reset to be ";", then (SETSYNTAX '; '* FILERDTBL) should be performed, and (; . PRINTCOMMENT) added to PRETTYPRINTMACROS.

6.8.3 Converting Comments to Lower Case

This section is for users operating on terminals without lower case, e.g. model 33 teletypes, who nevertheless would like their comments to be converted to lower case for more readable line-printer listings. If the second atom in a comment is %%, the text of the comment is converted to lower case so that it looks like English instead of LISP. Note that comments are converted *only* when they are actually written to a file by PRETTYPRINT.

The algorithm for conversion to lower case is the following: If the first character in an atom is ↑, do not change the atom (but remove the ↑). If the first character is %, convert the atom to lower case.[16] If the atom (minus any trailing punctuation marks) is an Interlisp word,[17] do not change it. Otherwise, convert the atom to lower case. Conversion only affects the upper case alphabet, i.e., atoms already converted to lower case are not changed if the comment is converted again. When converting, the first character in the comment and the first character following each period are left capitalized. After conversion, the comment is physically modified to be the lower case text minus the %% flag, so that conversion is thus

[15] Unless it believes the expression beginning with * is not actually a comment, e.g., if the next atom is "." or E.

[16] User must type %% as % is the escape character.

[17] i.e., is a bound or free variable for the function containing the comment, or has a top level value, or is a defined function, or has a non-NIL property list.

only performed once (unless the user edits the comment inserting additional upper case text and another %% flag).

LCASELST [Variable]

Words on LCASELST will always be converted to lower case. LCASELST is initialized to contain words which are Interlisp functions but also appear frequently in comments as English words (AND, EVERY, GET, GO, LAST, LENGTH, LIST, etc.). Therefore, if one wished to type a comment including the lisp fuction GO, it would be necessary to type ↑GO in order that it might be left in upper case.

UCASELST [Variable]

Words on UCASELST (that do not appear on LCASELST) will be left in upper case. UCASELST is initialized to NIL.

ABBREVLST [Variable]

ABBREVLST is used to distinguish between abbreviations and words that end in periods. Normally, words that end in periods and occur more than halfway to the right margin cause carriage-returns. Furthermore, during conversion to lowercase, words ending in periods, except for those on ABBREVLST, cause the first character in the *next* word to be capitalized. ABBREVLST is initialized to the upper and lower case forms of ETC., I.E., and E.G..

6.8.4 Special Prettyprint Controls

PRETTYTABFLG [Variable]

In order to save space on files, tabs are used instead of spaces for the inital spaces on each line, assuming that each tab corresponds to 8 spaces. This results in a reduction of file size by about 30%. Tabs are not used if PRETTYTABFLG is set to NIL (initially T).

#RPARS [Variable]

Controls the number of right parentheses necessary for square bracketing to occur. If #RPARS=NIL, no brackets are used. #RPARS is initialized to 4.

FIRSTCOL [Variable]

The starting column for comments. Initial setting is 48. Comments run between FIRSTCOL and LINELENGTH. If a word in a comment ends with a "." and is not on the list ABBREVLST, and the position is greater than halfway between FIRSTCOL and LINELENGTH, the next word in the comment begins on a new line. Also, if a list is encountered in a comment, and the position is greater than halfway, the list begins on a new line.

PRETTYLCOM [Variable]

If a comment is bigger (using COUNT) than PRETTYLCOM in size, it is printed starting at column 10, instead of FIRSTCOL. PRETTYLCOM is initialized to 14 (arrived at empirically). Comments are also printed starting at column 10 if their second element is also a *, i.e., comments of the form (* * --).

#CAREFULCOLUMNS [Variable]

In the interests of efficiency, PRETTYPRINT approximates the number of characters in each atom, rather than calling NCHARS, when computing how much will fit on a line. This procedure works satisfactorily in most cases. However, users with unusually long atoms in their programs, e.g., such as produced by CLISPIFY, may occasionlly encounter some glitches in the output produced by PRETTYPRINT. The value of #CAREFULCOLUMNS tells PRETTYPRINT how many columns (counting from the right hand margin) in which to actually compute NCHARS instead of approximating. Setting #CAREFULCOLUMNS to 20 or 30 will eliminate the glitches, although it will slow down PRETTYPRINT slightly. #CAREFULCOLUMNS is initially 0.

(WIDEPAPER *FLG*) [Function]

(WIDEPAPER T) sets FILELINELENGTH to 120, FIRSTCOL to 80, and PRETTYLCOM to 28. These are useful settings for prettyprinting files to be listed on wide paper. (WIDEPAPER) restores these parameters to their initial values. The value of WIDEPAPER is its previous setting.

PRETTYFLG [Variable]

If PRETTYFLG is NIL, PRINTDEF uses PRIN2 instead of prettyprinting. This is useful for producing a fast symbolic dump (see FAST option of MAKEFILE, page 11.6). Note that the file loads the same as if it were prettyprinted. PRETTYFLG is initially set to T. PRETTYFLG should not be set to NIL if comment pointers (page 6.51) are being used.

CLISPIFYPRETTYFLG [Variable]

Used to inform PRETTYPRINT to call CLISPIFY on selected function definitions before printing them (see page 16.20).

PRETTYPRINTMACROS [Variable]

An association-list that enables the user to control the formatting of selected expressions. CAR of each expression being PRETTYPRINTed is looked up on PRETTYPRINTMACROS, and if found, CDR of the corresponding entry is applied to the expression. If the result of this application is NIL, PRETTYPRINT ignores the expression; i.e., it prints nothing, assuming that the prettyprintmacro has done any desired printing. If the result of applying the prettyprint macro is non-NIL, the result is prettyprinted in the normal fashion. This gives the user the option of computing some other expression to be prettyprinted in its place. PRETTYPRINTMACROS is initially NIL.

Note: "prettyprinted in the normal fashion" includes processing prettyprint macros, unless the prettyprint macro returns a structure EQ to the one it was handed, in which case the potential recursion is broken.

PRETTYPRINTYPEMACROS [Variable]

A list of elements of the form (*TYPENAME* . *FN*). For types other than lists and atoms, the type name of each datum to be prettyprinted is looked up on PRETTYPRINTYPEMACROS, and if found, the corresponding function is applied to the datum about to be printed, instead of simply printing it with PRIN2. PRETTYPRINTYPEMACROS is initially NIL.

PRETTYEQUIVLST [Variable]

An association-list that tells PRETTYPRINT to treat a CAR-of-form the same as some other CAR-of-form. For example, if (QLAMBDA . LAMBDA) appears on PRETTYEQUIVLST, then expressions beginning with QLAMBDA are prettyprinted the same as LAMBDAs. PRETTYEQUIVLST is initially NIL. Currently, PRETTYEQUIVLST only allows (i.e., supports in an interesting way) equivalences to forms that PRETTYPRINT internally handles. Equivalence to forms for which the user has specified a prettyprint macro should be made by adding further entries to PRETTYPRINTMACROS

CHANGECHAR [Variable]

If non-NIL, and PRETTYPRINT is printing to a file or display terminal, PRETTYPRINT prints CHANGECHAR in the right hand margin while printing those expressions marked by the editor as having been changed (see page 17.22). CHANGECHAR is initially |.

6.8.5 Font Package

PRETTYPRINT contains a facility for printing elements of various classes (user functions, system functions, clisp words, comments, etc.) in different fonts to emphasize (or deemphasize) their importance, and in general to provide for more pleasing printout when printing to a file. Of course, in order to be useful, this facility requires that the user has access to a printer which supports multiple fonts, such as an XGP.

Prettyprint signals font changes by inserting a user-defined escape sequence, e.g. ↑F↑C meaning change to font 3, ↑F↑A change back to font 1, etc. It is convenient if these sequences can consist of control characters, because by making these characters be separator charactors in FILERDTBL, a file with font changes in it can also be loaded back in. Otherwise, the user would have to dump two files, one for listing, and one for loading.

Currently, the user can specify fonts for each of the following eight classes, each different, or the same for several classes.

LAMBDAFONT The font for printing the name of the function being prettyprinted, before the actual definition (usually a large font).

CLISPFONT If CLISPFLG is on, the font for printing any clisp words, i.e. atoms with property CLISPWORD.

COMMENTFONT The font for everything inside of a comment.

USERFONT The font for the name of any function in the file, or any member of the list FONTFNS.

SYSTEMFONT The font for any other (defined) function.

CHANGEFONT The font for anything in an expression marked by the editor as having been changed.

PRETTYCOMFONT The font used in printing the operand of a file package command.

DEFAULTFONT The font for everything else, or any of the above classes for which a font is not specified.

Note: the output primitives PRINT, PRIN1, etc., currently do not know about variable width fonts, so the user may have to experiment to find a compatible (pleasing) set of fonts. Note also that the user does not set LAMBDAFONT, CLISPFONT, et al, but indicates what font to be used by including an appropriate entry in FONTPROFILE. FONTSET will then set LAMBDAFONT, CLISPFONT, et al, to a data structure that contains the necessary information for performing the font change.

FONTPROFILE [Variable]

A list of elements of the form (*FONTCLASS* NIL *FONT#*),[18] where *FONTCLASS* is one of the eight font classes and *FONT#* is the font number for that class. it is assumed that the user has some way of communicating to the printing device the correspondence between font numbers and fonts. For each fontclass, the escape sequence consists of FONTESCAPECHAR followed by the character *code* for the font number, i.e. for font number 1, ↑A, for font number 2, ↑B, etc.

If *FONT#* is NIL for any fontclass, the DEFAULTFONT is used. Note that the DEFAULTFONT must be specified or an error is generated.

The operation of the font package is affected by a large number of parameters, e.g. FILELINELENGTH, LISTFILESTR, etc. plus the various fontnames themselves. To facilitate switching back and forth between various configurations, the font package allows the user to set the various parameters to their desired values, and then use the function FONTNAME to package up and save this configuration. Subsequently, the user invokes this configuration by performing (FONTSET *NAME*).

Note that the user may also want to reset FILELINELENGTH (page 23.14), PRETTYLCOM (page 6.53), and FIRSTCOL (page 6.53) as a part of various font configurations.

(FONTNAME *NAME*) [Function]

Performs some processing on FONTPROFILE, and then collects names and values of variables on FONTDEFSVARS, and saves them on FONTDEFS.

(FONTSET *NAME*) [Function]

Restores font configuration for *NAME*. Generates an error if *NAME* not previously defined.

FONTDEFSVARS [Variable]

The list of variables to be packaged by a FONTNAME. Initially FONTCHANGEFLG, FILELINELENGTH, COMMENTLINELENGTH, FIRSTCOL, PRETTYLCOM, LISTFILESTR, and FONTPROFILE.

FONTESCAPECHAR [Variable]

The character or string used to signal the start of a font escape sequence.

FONTCHANGEFLG [Variable]

If T, enables fonts, if NIL, disables fonts, i.e. no font changes are performed when prettyprinting.

[18] The NIL is a place marker. FONTNAME replaces (RPLACA) CADR when the font configuration is defined.

LISTFILESTR [Variable]

Passed to the operating system by LISTFILES (page 11.9). Can be used to specify subcommands to the LIST command, e.g. to establish correspondance between font number and font name.

COMMENTLINELENGTH [Variable]

Since comments are usually printed in a smaller font, COMMENTLINELENGTH is provided to offset the fact that Interlisp does not know about font widths. When FONTCHANGEFLG=T, CAR of COMMENTLINELENGTH is the linelength used to print short comments, i.e. those printed in the right margin, and CDR is the linelength used when printing full width comments.

(CHANGEFONT *FONTCLASS*) [Function]

Prints the font escape sequence to change to *FONTCLASS*. Note that *FONTCLASS* is not a font name, so one should use (CHANGEFONT LAMBDAFONT), not (CHANGEFONT 'LAMBDAFONT). For use in PRETTYPRINTMACROS.

FONTDEFS [Variable]

The dictionary of font configurations. FONTDEFS is a list of elements of form (*NAME* . *PARAMETER-PAIRS*). To save a configuration on a file after performing a FONTNAME to define it, the user could either save the entire value of FONTDEFS, or simply use an ALISTS file package command (page 11.23) to dump out just the one configuration.

6.9 ASKUSER

DWIM, the compiler, the editor, and many other system packages all use ASKUSER, an extremely general user interaction package, for their interactions with the user at the terminal. ASKUSER takes as its principal argument *KEYLST* which is used to drive the interaction. *KEYLST* specifies what the user can type at any given point, how ASKUSER should respond to the various inputs, what value should be returned by ASKUSER, and is also used to present the user at any given point with a list of the possible responses. ASKUSER also takes other arguments which permit specifying a wait time, a default value, a message to be printed on entry, a flag indicating whether or not typeahead is to be permitted, a flag indicating whether the transaction is to be stored on the history list (page 8.1), a default set of options, and an (optional) input file/string.

6.9.1 Startup Protocol

Interlisp permits and encourages the user to typeahead; in actual practice, the user frequently does this. This presents a problem for ASKUSER. When ASKUSER is entered and there has been typeahead, was the input intended for ASKUSER, or was the interaction unanticipated, and the user simply typing ahead to some other program, e.g. the programmer's assistant? Even where there was no typeahead, i.e., the user starts typing *after* the call to ASKUSER, the question remains of whether the user had time to see the message from ASKUSER and react to it, or simply began typing ahead at an inauspicious moment. Thus, what is needed is an interlock mechanism which warns the user to stop typing, gives him a chance to respond to the warning, and then allows him to begin typing to ASKUSER.

Therefore, when ASKUSER is first entered, and the interaction is to take place with a terminal, and typeahead to ASKUSER is not permitted, the following protocol is observed:

(1) If there is typeahead, ASKUSER clears and saves the input buffers and rings the bell to warn the user to stop typing. The buffers will be restored when ASKUSER completes operation and returns.

(2) If *MESS*, the message to be printed on entry, is not NIL (the typical case), ASKUSER then prints *MESS* if it is a string, otherwise CAR of *MESS*, if *MESS* is a list.

(3) After printing *MESS* or CAR of *MESS*, ASKUSER waits until the output has actually been printed on the terminal to make sure that the user has actually had a chance to see the output. This also give the user a chance to react. ASKUSER then checks to see if anything additional has been typed in the intervening period since it first warned the user in (1). If something has been typed, ASKUSER clears it out and again rings the bell. This latter material, i.e., that typed between the entry to ASKUSER and this point, is discarded and will not be restored since it is not certain whether the user simply reacted quickly to the first warning (bell) and this input is intended for ASKUSER, or whether the user was in the process of typing ahead when the call to ASKUSER occurred, and did not stop typing at the first warning, and therefore this input is a continuation of input intended for another program.

Anything typed after (3) is considered to be intended for ASKUSER, i.e., once the user sees *MESS* or CAR of *MESS*, he is free to respond. For example, UNDO (page 8.11) calls ASKUSER when the number of undosaves are exceeded for an event with *MESS*=(LIST *NUMBER-UNDOSAVES* "undosaves, continue saving"). Thus, the user can type a response as soon as *NUMBER-UNDOSAVES* is typed.

(4) ASKUSER then types the rest of *MESS*, if any.

(5) Then ASKUSER goes into a wait loop until something is typed. If *WAIT*, the wait time, is not NIL, and nothing is typed in *WAIT* seconds, ASKUSER will type "..." and treat the elements of *DEFAULT*, the default value, as a list of characters, and begin processing them exactly as though they had been typed. If the user does type anything within *WAIT* seconds, he can then wait as long as he likes, i.e., once something has been typed, ASKUSER will not use the default value specified in *DEFAULT*.

If the user wants to consider his response for more than *WAIT* seconds, and does not want ASKUSER to default, he can type a carriage return or a space, which are ignored if they are not specified as acceptable inputs by *KEYLST* (see below) and they are the first thing typed.

If the calling program knows that the user is expecting an interaction with ASKUSER, e.g. another interaction preceded this one, it can specify in the call to ASKUSER that typeahead is permitted. In this case, ASKUSER simply notes whether there is any typeahead,[19] then prints *MESS* and goes into a wait loop as described above.

(6) Finally, if the interaction is not with the terminal, i.e., the optional input file/string is specified, ASKUSER simply prints *MESS* and begins reading from the file/string.

[19] In this case, if the typeahead turns out to contain unacceptable input, ASKUSER will assume that the typeahead was not intended for ASKUSER, and will restore the typeahead when it completes operation and returns.

6.9.2 Operation

All input operations are executed with the terminal table in the variable ASKUSERTTBL,, in which (1) (CONTROL T) has been executed, so that ASKUSER can interact with the user after each character is typed; and (2) (ECHOMODE NIL) has been executed, so that ASKUSER can decide *after* it reads a character whether or not the character should be echoed, and with what, e.g. unacceptable inputs are never echoed.

As each character is typed, it is matched against *KEYLST*, and appropriate echoing and/or prompting is performed. If the user types an unacceptable character, ASKUSER simply rings the bell and allows him to try again.

At any point, the user can type ? and receive a list of acceptable responses at that point (generated from *KEYLST*), or type a control-A, control-Q, control-X, or <del>, which causes ASKUSER to reinitialize, and start over.

Note that ?, Control-A, Control-Q, and Control-X will not work if they are acceptable inputs, i.e., they match one of the keys on *KEYLST*. <del> will not work if it is an interrupt character, in which case it is not seen by ASKUSER.

When an acceptable sequence is completed, ASKUSER returns the indicated value.

6.9.3 Format of KEYLST

KEYLST is a list of elements of the form (*KEY PROMPTSTRING* . *OPTIONS*), where *KEY* is an atom or a string (equivalent), *PROMPTSTRING* is an atom or a string, and *OPTIONS* a list of options in property list format. The following options are recognized and explained below: KEYLST, CONFIRMFLG, PROMPTCONFIRMFLG, NOCASEFLG, RETURN, EXPLAINSTRING, NOECHOFLG, KEYSTRING, PROMPTON, COMPLETEON, AUTOCOMPLETEFLG. If an option is specified in *OPTIONS*, the value of the option is the next element. Otherwise, if the option is specified in *OPTIONSLST* (the seventh argument to ASKUSER), its value is the next element on *OPTIONSLST*. Thus, *OPTIONSLST* can be used to provide default options for an entire *KEYLST*, rather than having to include the option at each level. If an option does not appear on either *OPTIONS* or *OPTIONSLST*, its value is NIL.

For convenience, an entry on *KEYLST* of the form (*KEY* . *ATOM/STRING*), can be used as an abbreviation for (*KEY ATOM/STRING* CONFIRMFLG T), and an entry of just the form *KEY*, i.e., a non-list, as an abbreviation for (*KEY* NIL CONFIRMFLG T).

As each character is read, it is matched against the currently active keys. A character matches a key if it is the same character as that in the corresponding position in the key, or, if the character is an alphabetic character, if the characters are the same without regard for upper/lower case differences, i.e. "A" matches "a" and vice versa.[20] In other words, if two characters have already been input and matched, the third character is matched with each active key by comparing it with the third character of that key. If the character matches with one or more of the keys, the entries on *KEYLST* corresponding to the remaining keys are discarded. If the character does not match with any of the keys, the character is not echoed, and a bell is rung instead.

[20] Unless the NOCASEFLG option (page 6.62) is T.

When a key is complete, *PROMPTSTRING* is printed (NIL is equivalent to "", the empty string, i.e., nothing will be printed). Then, if the value of the CONFIRMFLG option is T, ASKUSER waits for confirmation of the key by a^{cr}[21] or space. Otherwise, the key does not require confirmation.

Then, if the value of the KEYLST option is not NIL, its value becomes the new *KEYLST*, and the process recurses. Otherwise, the key is a "leaf," i.e., it terminates a particular path through the original, top-level *KEYLST*, and ASKUSER returns the result of packing all the keys that have been matched and completed along the way (unless the RETURN option is used to specify some other value, as described below).

For example, the following *KEYLST* is the default *KEYLST*, i.e., is used when ASKUSER is called with *KEYLST*=NIL: ((Y "es^{cr}") (N "o^{cr}"))

This *KEYLST* specifies that if (as soon as) the user types Y (or y), ASKUSER echoes with Y, prompts with "es^{cr}", and returns Y as its value. Similarly, if the user types N, ASKUSER echoes the N, prompts with "o^{cr}", and returns N. If the user types ?, ASKUSER prints:

```
Yes
No
```

to indicate his possible responses. All other inputs are unacceptable, and ASKUSER will ring the bell and not echo or print anything.

Here is a more complicated example, the *KEYLST* used for the compiler questions (page 12.1):

```
((ST "ore and redefine " KEYLST ("" (F . "orget exprs"))
 (S . "ame as last time")
 (F . "File only")
 (T . "o terminal")
 1
 2
 (Y . "es")
 (N . "o"))
```

When ASKUSER is called with this *KEYLST*, and the user types an S, two keys are matched: ST and S. The user can then type a T, which matches only the ST key, or confirm the S key by typing a^{cr} or space. If the user confirms the S key, ASKUSER prompts with "ame as last time", and returns S as its value. (Note that the confirming character is not included in the value.) If the user types a T, ASKUSER prompts with "ore and redefine", and makes ("" (F . "orget exprs")) be the new *KEYLST*, and waits for more input. The user can then type an F, or confirm the "" (which essentially starts out with all of its characters matched). If he confirms the "", ASKUSER returns ST as its value the result of packing ST and "". If he types F, ASKUSER prompts with "orget exprs", and waits for confirmation again. If the user then confirms, ASKUSER returns STF, the result of packing ST and F.

As mentioned earlier, at any point the user can type a ? and be prompted with the possible responses. For example, if the user types S and then ?, ASKUSER will type:

```
STore and redefine Forget exprs
STore and redefine
Same as last time
```

[21] cr is used throughout the discussion to denote carriage return.

6.9.4 Completing a Key

The decision about when a key is complete is more complicated than simply whether or not all of its characters have been matched. In the example above, all of the characters in the S key are matched as soon as the S has been typed, but until the next character is typed, ASKUSER does not know whether the S completes the S key, or is simply the first character in the ST key. Therefore, a key is considered to be complete when:

(1) All of its characters have been matched and it is the only key left, i.e., there are no other keys for which this key is a substring; or

(2) All of its characters have been matched and a confirming character is typed; or

(3) All of its characters have been matched, and the value of the CONFIRMFLG option is NIL, and the value of the KEYLST option is not NIL, and the next character matches one of the keys on the value of the KEYLST option; or

(4) There is only one key left and a confirming character is typed. Note that if the value of CONFIRMFLG is T, the key still has to be confirmed, regardless of whether or not it is complete. For example, if the first entry in the above example were instead

```
(ST "ore and redefine " CONFIRMFLG T KEYLST ("" (F . "orget exprs"))
```

and the user wanted to specify the STF path, he would have to type ST, *then* confirm before typing F, even though the ST completed the ST key by the rule in case (1). However, he would be prompted with "ore and redefine" as soon as he typed the T, and completed the ST key.

Case (2) says that confirmation can be used to complete a key in the case where it is a substring of another key, even where the value of CONFIRMFLG is NIL. In this case, the confirming character doubles as both an indicator that the key is complete, and also to confirm it, if necessary. This situation corresponds to typing S^{cr} in the above example.

Case (3) says that if there were another entry whose key was STX in the above example, so that after the user typed ST, two keys, ST and STX, were still active, then typing F would complete the ST key, because F matches the (F . "orget exprs") entry on the value of the KEYLST option of the ST entry. In this case, "ore and redefine" would be printed *before* the F was echoed.

Finally, case (4) says that the user can use confirmation to specify completion when only one key is left, even when all of its characters have not been matched. For example, if the first key in the above example were STORE, the user could type ST and then confirm, and ORE would be echoed, followed by whatever prompting was specified. In this case, the confirming character also confirms the key if necessary, so that no further action is required, even when the value of CONFIRMFLG is T.

Case (4) permits the user not to have to type every character in a key when the key is the only one left. Even when there are several active keys, the user can type type $ (the ESC key, or on some terminals, the key labelled ALT) to specify the next N>0 common characters among the currently active keys. The effect is exactly the same as though these characters had been typed. If there are no common characters in the active keys at that point, i.e. $N=0$, the $ is treated as an incorrect input, and the bell is rung. For example, if *KEYLST* is (CLISPFLG CLISPIFYPACKFLG CLISPIFTRANFLG), and the user types C followed by $, ASKUSER will supply the L, I, S, and P. The user can then type F followed by^{cr} or space to complete and confirm CLISPFLG, as per case (4), or type I, followed by $, and ASKUSER will supply the F, etc. Note that the characters supplied do not have to correspond to a terminal segment of any of the keys. Note also that the $ does not confirm the key, although it may complete it in the case that there is only one key active.

If the user types a confirming character when several keys are left, the next *N*>0 common characters are still supplied, the same as with $. However, ASKUSER assumes the intent was to complete a key, i.e., case (4) is being invoked. Therefore, after supplying the next *N* characters, the bell is rung to indicate that the operation was not completed. In other words, typing a confirming character has the same effect as typing an $ in that the next *N* common characters are supplied. Then, if there is only one key left, the key is complete (case 4) and confirmation is not required. If the key is not the only key left, the bell is rung.

6.9.5 Options

KEYLST
When a key is complete, if the value of the KEYLST option is not NIL, this value becomes the new *KEYLST* and the process recurses. Otherwise, the key terminates a path through the original, top-level *KEYLST*, and ASKUSER returns the indicated value.

CONFIRMFLG
If T, the key must be confirmed with either a^{cr} or a space. If the value of CONFIRMFLG is a *list*, the confirming character may be any member of the list.

PROMPTCONFIRMFLG
If T, whenever confirmation is required, the user is prompted with the string " [confirm] ".

NOCASEFLG
If T, says do *not* perform case independent matching on alphabetic characters. If NIL, do perform case independent matching, i.e. "A" matches with "a" and vice versa.

RETURN
If non-NIL, EVAL of the value of the RETURN option is returned as the value of ASKUSER. Note that different RETURN options can be specified for different keys. The variable ANSWER is bound in ASKUSER to the list of keys that have been matched. In other words, RETURN (PACK ANSWER) would be equivalent to what ASKUSER normally does.

EXPLAINSTRING
If the value of the EXPLAINSTRING option is non-NIL, its value is printed when the user types a ?, rather than *KEY* + *PROMPTSTRING*. EXPLAINSTRING enables more elaborate explanations in response to a ? than what the user sees when he is prompted as a result of simply completing keys. See example below.

NOECHOFLG
If non-NIL, characters that are matched (or automatically supplied as a result of typing $ or confirming) are not echoed, nor is the confirming character, if any. The value of NOECHOFLG is automatically NIL when ASKUSER is reading from a file or string. The decision about whether or not to echo a character that matches several keys is determined by the value of the NOECHOFLG option for the first key.

Example: one of the entries on the *KEYLST* used by ADDTOFILES? (page 11.8) is:

```
(] "Nowhere$^{cr}$" NOECHOFLG T
   EXPLAINSTRING "] - nowhere, item is marked as a dummy$^{cr}$")
```

When the user types], ASKUSER just prints "Nowhere^{cr}", i.e., the] is not echoed. If the user types ?, the explanation corresponding to this entry will be:

```
] - nowhere, item is marked as a dummy
```

KEYSTRING — If non-NIL, characters that are matched are echoed as though the value of KEYSTRING were used in place of the key. KEYSTRING is also used for computing the value returned. The main reason for this feature is to enable echoing in lowercase.

PROMPTON — If non-NIL, *PROMPTSTRING* is printed *only* when the key is confirmed with a member of the value of PROMPTON. See example below.

COMPLETEON — When a confirming character is typed, the *N* characters that are automatically supplied, as specified in case (4), are echoed *only* when the key is confirmed with a member of the value of PROMPTON.

The PROMPTON and COMPLETEON options enable the user to construct a *KEYLST* which will cause ASKUSER to emulate the action of the TENEX exec. The protocol followed by the TENEX exec is that the user can type as many characters as he likes in specifying a command. The command can be completed with a^{cr} or space, in which case no further output is forthcoming, or with a $, in which case the rest of the characters in the command are echoed, followed by some prompting information. The following *KEYLST* would handle the TENEX COPY and CONNECT comands:

```
((COPY " (FILE LIST) "
         PROMPTON ($)
         COMPLETEON ($)
         CONFIRMFLG ($))
 (CONNECT " (TO DIRECTORY) "
         PROMPTON ($)
         COMPLETEON ($)
         CONFIRMFLG ($)))
```

AUTOCOMPLETEFLG

If the value of the AUTOCOMPLETEFLG option is not NIL, ASKUSER will automatically supply unambiguous characters whenever it can, i.e., ASKUSER acts as though $ were typed after each character (except that it does not ring the bell if there are no unambiguous characters).

MACROCHARS — value is a list of dotted pairs of form (*CHARACTER* . *FORM*). When *CHARACTER* is typed, and it does not match any of the current keys, *FORM* is evaluated and nothing else happens, i.e. the matching process stays where it is. For example, ? could have been implemented using this option. Essentially MACROCHARS provides a read macro facility while inside of ASKUSER (since ASKUSER does READC's, read macros defined via the readtable are never invoked).

EXPLAINDELIMITER

value is what is printed to delimit explanation in response to ?. Initially "^{cr}" but can be reset, e.g. to ", ", for more linear output.

6.9.6 Special Keys

& can be used as a key to match with any single character, provided the character does not match with some other key at that level. For the purposes of echoing and returning a value, the effect is the same as though the character that were matched actually appeared as the key.

$ (esc) can be used as a key to match with the result of a single call to READ. For example, if the first entry in the TENEX *KEYLST* above were:

```
(COPY " (FILE LIST) "
          PROMPTON ($)
          COMPLETEON ($)
          CONFIRMFLG ($)
          KEYLST (($ NIL RETURN ANSWER)))
```

then if the user typed COP FOO^{cr}, (COPY FOO) would be returned as the value of ASKUSER. One advantage of using $, rather than having the calling program perform the READ, is that the call to READ from inside ASKUSER is ERRORSET protected, so that the user can back out of this path and reinitialize ASKUSER, e.g. to change from a COPY command to a CONNECT command, simply by typing control-E.

$$ can be used as a key to match with the result of a single call to READLINE.

A list can be used as a key, in which case the list/form is evaluated and its value "matches" the key. This feature is provided primarily as an escape hatch for including arbitrary input operations as part of an ASKUSER sequence. For example, the effect of $$ could be achieved simply by using (READLINE T) as a key.[22]

"" can be used as a key. Since it has no characters, all of its characters are automatically matched. "" essentially functions as a place marker. For example, one of the entries on the *KEYLST* used by ADDTOFILES? is:

```
("" "File/list:  "
     EXPLAINSTRING "a file name or name of a function list"
     KEYLST ($))
```

Thus, if the user types a character that does not match any of the other keys on the *KEYLST*, then the character completes the "" key, by virtue of case (4), since the character *will* match with the $ in the inner *KEYLST*. ASKUSER then prints "File/list: " *before* echoing the character, then calls READ. The character will be read as part of the READ. The value returned by ASKUSER will be the value of the READ.

(ASKUSER *WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG OPTIONSLST FILE*) [Function]

WAIT is either NIL or a number (of seconds). *DEFAULT* is a single character or a sequence (list) of characters to be used as the default inputs for the case when *WAIT* is not NIL and more than *WAIT* seconds elapse without any input. In this

[22] For $, $$, or a list, if the last character read by the input operation is a separator, the character is treated as a confirming character for the key. However, if the last character is a break character, it will be matched against the next key.

case, the character(s) from *DEFAULT* are processed exactly as though they had been typed, except that ASKUSER first types "...".

MESS is the initial message to be printed by ASKUSER, if any, and can be a string, or a list. In the latter case, each element of the list is printed, separated by spaces, and terminated with a " ? ". *KEYLST* and *OPTIONSLST* were described earlier. *TYPEAHEAD* is T if the user is permitted to typeahead a response to ASKUSER. NIL means any typeahead should be cleared and saved. *LISPXPRNTFLG* determines whether or not the interaction is to be recorded on the history list. *FILE* can be either NIL (in which case it is set to T), the name of a file, or a string.[23] All input operations take place from *FILE* until an unacceptable input is encountered, i.e., one that does not conform to the protocol defined by *KEYLST*. At that point, *FILE* is set to T, *DEFAULT* is set to NIL, the input buffer is cleared, and a bell is rung. Unacceptable inputs are not echoed.

The value of ASKUSER is the result of packing all the keys that were matched, unless the RETURN option is specified (page 6.62).

(MAKEKEYLST *LST DEFAULTKEY LCASEFLG —*) [Function]

LST is a list of atoms or strings. MAKEKEYLST returns an ASKUSER *KEYLST* which will permit the user to specify one of the elements on *LST* by either typing enough characters to make the choice unambiguous, or else typing a number between 1 and *N*, where *N* is the length of *LST*.

For example, if ASKUSER is called with *KEYLST* = (MAKEKEYLST '(CONNECT SUPPORT COMPILE)), then the user can type C-O-N, S, C-O-M, 1, 2, or 3 to indicate one of the three choices.

If *LCASEFLG*=T, then echoing of upper case elements will be in lower case (but the value returned will still be one of the elements of *LST*). If *DEFAULTKEY* is non-NIL, it will be the last key on the *KEYLST*. Otherwise, a key which permits the user to indicate "No - none of the above" choices, in which case the value returned by ASKUSER will be NIL.

[23] If *FILE* is a string, and all of its characters are read before ASKUSER finishes, *FILE* will be reset to T, and the interaction will continue with ASKUSER reading from the terminal.

CHAPTER 7

VARIABLE BINDINGS AND THE INTERLISP STACK

A number of schemes have been used in different implementations of LISP for storing the values of variables. These include:

(1) Storing values on an association list paired with the variable names.

(2) Storing values on the property list of the atom which is the name of the variable.

(3) Storing values in a special value cell associated with the atom name, putting old values on a pushdown list, and restoring these values when exiting from a function.

(4) Storing values on a pushdown list.

Interlisp-10 uses the third scheme, so called "shallow binding". When a function is entered, the value of each variable bound by the function (function argument) is stored in a value cell associated with that variable name. The value that was in the value cell is stored in a block of storage called the basic frame for this function call. In addition, on exit from the function each variable must be individually unbound; that is, the old value saved in the basic frame must be restored to the value cell. Thus there is a higher cost for binding and unbinding a variable than in the fourth scheme, "deep binding". However, to find the current value of any variable, it is only necessary to access the variable's value cell, thus making variable reference considerably cheaper under shallow binding than under deep binding, especially for free variables. However, the shallow binding scheme used does require an additional overhead in switching contexts when doing "spaghetti stack" operations.

Interlisp-D uses the forth scheme, "deep binding." Every time a function is entered, a basic frame containing the new variables is put on top of the stack. Therefore, any variable reference requires searching the stack for the first instance of that variable, which makes free variable use somewhat more expensive than in a shallow binding scheme. On the other hand, spaghetti stack operations are considerably faster. Some other tricks involving copying freely-referenced variables to higher frames on the stack are also used to speed up the search.

The basic frames are allocated on a stack or pushdown list; for most user purposes, these frames should be thought of as containing the variable names associated with the function call, and the *current* values for that frame. The descriptions of the stack functions in below are presented from this viewpoint. Both interpreted and compiled functions store both the names and values of variables so that interpreted and compiled functions are compatible and can be freely intermixed, i.e., free variables can be used with no SPECVAR declarations necessary. However, it is possible to *suppress* storing of names in compiled functions, either for efficiency or to avoid a clash, via a LOCALVAR declaration (see page 12.4). The names are also very useful in debugging, for they make possible a complete symbolic backtrace in case of error.

In addition to the binding information, additional information is associated with each function call: access information indicating the path to search the basic frames for variable bindings, control information, and temporary results are also stored on the stack in a block called the frame extension. The interpreter also stores information about partially evaluated expressions as described on page 7.10.

7.1 THE SPAGHETTI STACK

The Bobrow/Wegbreit paper, "A Model and Stack Implementation for Multiple Environments",[1] describes an access and control mechanism more general than the simple pushdown stack. The access and control mechanism used by Interlisp is a slightly modified version of the one proposed by Bobrow and Wegbreit. This mechanism is called the "spaghetti stack."

The spaghetti system presents the access and control stack as a data structure composed of "frames." The functions described below operate on this structure. These primitives allow user functions to manipulate the stack in a machine independent way. Backtracking, coroutines, and more sophisticated control schemes can be easily implemented with these primitives.

The evaluation of a function requires the allocation of storage to hold the values of its local variables during the computation. In addition to variable bindings, an activation of a function requires a return link (indicating where control is to go after the completion of the computation) and room for temporaries needed during the computation. In the spaghetti system, one "stack" is used for storing all this information, but it is best to view this stack as a tree of linked objects called frame extensions (or simply frames).

A frame extension is a variable sized block of storage containing a frame name, a pointer to some variable bindings (the BLINK), and two pointers to other frame extensions (the ALINK and CLINK). In addition to these components, a frame extension contains other information (such as temporaries and reference counts) that does not interest us here.

The block of storage holding the variable bindings is called a basic frame. A basic frame is essentially an array of pairs, each of which contains a variable name and its value. The reason frame extensions point to basic frames (rather than just having them "built in") is so that two frame extensions can share a common basic frame. This allows two processes to communicate via shared variable bindings.

The chain of frame extensions which can be reached via the successive ALINKs from a given frame is called the "access chain" of the frame. The first frame in the access chain is the starting frame. The chain through successive CLINKs is called the "control chain".

A frame extension completely specifies the variable bindings and control information necessary for the evaluation of a function. Whenever a function (or in fact, any form which generally binds local variables) is evaluated, it is associated with some frame extension.

In the beginning there is precisely one frame extension in existence. This is the frame in which the top-level call to the interpreter is being run. This frame is called the "top-level" frame.

Since precisely one function is being executed at any instant, exactly one frame is distinguished as having the "control bubble" in it. This frame is called the active frame. Initially, the top-level frame is the active frame. If the computation in the active frame invokes another function, a new basic frame and frame extension are built. The frame name of this basic frame will be the name of the function being called. The ALINK, BLINK, and CLINK of the new frame all depend on precisely how the function is invoked. The new function is then run in this new frame by passing control to that frame, i.e., it is made the active frame.

[1] *Communications of the ACM*, Vol. 16, 10, October 1973.

Once the active computation has been completed, control normally returns to the frame pointed to by the CLINK of the active frame. That is, the frame in the CLINK becomes the active frame.

In most cases, the storage associated with the basic frame and frame extension just abandoned can be reclaimed. However, it is possible to obtain a pointer to a frame extension and to "hold on" to this frame even after it has been exited. This pointer can be used later to run another computation in that environment, or even "continue" the exited computation.

A separate data type, called a stack pointer, is used for this purpose. A stack pointer is just a cell that literally points to a frame extension. Stack pointers print as *#ADR/FRAMENAME*, e.g., #1,13636/COND. Stack pointers are returned by many of the stack manipulating functions described below. Except for certain abbreviations (such as "the frame with such-and-such a name"), stack pointers are the only way the user can reference a frame extension. As long as the user has a stack pointer which references a frame extension, that frame extension (and all those that can be reached from it) will not be garbage collected.

Note that two stack pointers referencing the same frame extension are *not* necessarily EQ, i.e., (EQ (STKPOS 'FOO) (STKPOS 'FOO))=NIL. However, EQP can be used to test if two different stack pointers reference the same frame extension (page 2.3).

It is possible to evaluate a form with respect to an access chain other than the current one by using a stack pointer to refer to the head of the access chain desired. Note, however, that this can be very expensive when using a shallow binding scheme such as that in Interlisp-10. When evaluating the form, since all references to variables under the shallow binding scheme go through the variable's value cell, the values in the value cells must be adjusted to reflect the values appropriate to the desired access chain. This is done by changing all the bindings on the current access chain (all the name-value pairs) so that they contain the value current at the time of the call. Then along the new access path, all bindings are made to contain the previous value of the variable, and the current value is placed in the value cell. For that part of the access path which is shared by the old and new chain, no work has to be done. The context switching time, i.e. the overhead in switching from the current, active, access chain to another one, is directly proportional to the size of the two branches that are not shared between the access contexts. This cost should be remembered in using generators and coroutines (page 7.13).

7.2 STACK FUNCTIONS

In the descriptions of the stack functions below, when we refer to an argument as a stack descriptor, we mean that it is either a stack pointer or one of the following abbreviations:

- NIL means the active frame; that is, the frame of the stack function itself.

- T means the top-level frame.

- Any other literal atom is equivalent to (STKPOS *ATOM* -1).

- A number is equivalent to (STKNTH *NUMBER*).

In the stack functions described below, the following errors can occur: The error ILLEGAL STACK ARG occurs when a stack descriptor is expected and the supplied argument is either not a legal stack descriptor (i.e., not a stack pointer, litatom, or number), or is a litatom or number for which there is no corresponding stack frame, e.g., (STKNTH -1 'FOO) where there is no frame named FOO in the active control chain or (STKNTH -10 'EVALQT). The error STACK POINTER HAS BEEN RELEASED occurs whenever a released stack pointer is supplied as a stack descriptor argument for any purpose other than as a stack pointer to re-use.

Note: The creation of a single stack pointer can result in the retention of a large amount of stack space. Therefore, one should try to release stack pointers when they are no longer needed. See page 7.10.

(STKPOS *NAME N POS OLDPOS*) [Function]

Returns a stack pointer to the *N*th frame with frame name *NAME*. The search begins with (and includes) the frame specified by the stack descriptor *POS*. The search proceeds along the control chain from *POS* if *N* is negative, or along the access chain if *N* is positive. If *N* is NIL, -1 is used. Returns a stack pointer to the frame if such a frame exists, otherwise returns NIL. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is supplied and is a stack pointer and STKPOS returns NIL, *OLDPOS* is released. If *OLDPOS* is not a stack pointer it is ignored.

Note: (STKPOS 'STKPOS) causes an error, ILLEGAL STACK ARG; it is not permissible to create a stack pointer to the active frame.

(STKNTH *N POS OLDPOS*) [Function]

Returns a stack pointer to the *N*th frame back from the frame specified by the stack descriptor *POS*. If *N* is negative, the control chain from *POS* is followed. If *N* is positive the access chain is followed. If *N* equals 0, STKNTH returns a stack pointer to *POS* (this provides a way to copy a stack pointer). Returns NIL if there are fewer than *N* frames in the appropriate chain. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is not a stack pointer it is ignored.

Note: (STKNTH 0) causes an error, ILLEGAL STACK ARG; it is not possible to create a stack pointer to the active frame.

(STKNAME *POS*) [Function]

Returns the frame name of the frame specified by the stack descriptor *POS*.

(SETSTKNAME *POS NAME*) [Function]

Changes the frame name of the frame specified by *POS* to be *NAME*. Returns *NAME*.

(STKNTHNAME *N POS*) [Function]

Returns the frame name of the *N*th frame back from *POS*. Equivalent to (STKNAME (STKNTH *N POS*)) but avoids creation of a stack pointer.

In summary, STKPOS converts function names to stack pointers, STKNTH converts numbers to stack pointers, STKNAME converts stack pointers to function names, and STKNTHNAME converts numbers to function names.

(DUMMYFRAMEP *POS*) [Function]

Returns T if the user never wrote a call to the function at *POS*, e.g. in Interlisp-10, DUMMYFRAMEP is T for *PROG*LAM, *ENV*, and FOOBLOCK frames (see block compiler, page 12.13).

REALFRAMEP and REALSTKNTH can be used to write functions which manipulate the stack and work on either interpreted or compiled code:

(REALFRAMEP *POS INTERPFLG*) [Function]

Returns *POS* if *POS* is a "real" frame, i.e. if *POS* is not a dummy frame and *POS* is a frame that does not disappear when compiled (such as COND); otherwise NIL. If *INTERPFLG*=T, returns *POS* if *POS* is not a dummy frame. For example, if (STKNAME *POS*)=COND, (REALFRAMEP *POS*) is NIL, but (REALFRAMEP *POS* T) is *POS*.

(REALSTKNTH *N POS INTERPFLG OLDPOS*) [Function]

Returns a stack pointer to the *N*th (or -*N*th) frames for which (REALFRAMEP *POS INTERPFLG*) is *POS*.

The following functions are used for accessing and changing bindings. Some of functions take an argument, *N*, which specifies a particular binding in the basic frame. If *N* is a literal atom, it is assumed to be the name of a variable bound in the basic frame. If *N* is a number, it is assumed to reference the *N*th binding in the basic frame. The first binding is 1. If the basic frame contains no binding with the given name or if the number is too large or too small, the error ILLEGAL ARG occurs.

(STKSCAN *VAR IPOS OPOS*) [Function]

Searches beginning at *IPOS* for a frame in which a variable named *VAR* is bound. The search follows the access chain. Returns a stack pointer to the frame if found, otherwise returns NIL. If *OPOS* is a stack pointer it is reused, otherwise it is ignored.

(FRAMESCAN *ATOM POS*) [Function]

Returns the relative position of the binding of *ATOM* in the basic frame of *POS*. Returns NIL if *ATOM* is not found.

(STKARG *N POS* —) [Function]

Returns the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(STKARGNAME *N POS*) [Function]

Returns the name of the binding specified by *N*, in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(SETSTKARG *N POS VALUE*) [Function]

Sets the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns value.

(SETSTKARGNAME *N POS NAME*) [Function]

Sets the *NAME* of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *NAME*.

(STKNARGS *POS* —) [Function]

Returns the number of arguments bound in the basic frame of the frame specified by the stack descriptor *POS*.

(VARIABLES *POS*) [Function]

Returns a list of the variables bound at *POS*.

As an example of the use of STKNARGS and STKARGNAME, VARIABLES could be defined by:

```
(VARIABLES
   [LAMBDA (POS)
       (for N from 1 to (STKNARGS POS)
          collect (STKARGNAME N POS])
```

(STKARGS *POS* —) [Function]

Returns a list of the *values* of variables bound at *POS*.

The following functions are used to evaluate an expression in a different environment, and/or to alter the flow of control.

(ENVEVAL *FORM APOS CPOS AFLG CFLG*) [Function]

Evaluates *FORM* in the environment specified by *APOS* and *CPOS*. That is, a new active frame is created with the frame specified by the stack descriptor *APOS* as its ALINK, and the frame specified by the stack descriptor *CPOS* as its CLINK. Then *FORM* is evaluated. If *AFLG* is not NIL, and *APOS* is a stack pointer, then *APOS* will be released. Similarly, if *CFLG* is not NIL, and *CPOS* is a stack pointer, then *CPOS* will be released.

(ENVAPPLY *FN ARGS APOS CPOS AFLG CFLG*) [Function]

APPLYs *FN* to *ARGS* in the environment specified by *APOS* and *CPOS*. *AFLG* and *CFLG* have the same interpretation as with ENVEVAL.

(STKEVAL *POS FORM FLG* —) [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*. If *FLG* is not NIL and *POS* is a stack pointer, releases *POS*. The definition of STKEVAL is (ENVEVAL *FORM POS* NIL *FLG*).

(STKAPPLY *POS FN ARGS FLG* —) [Function]

Similar to STKEVAL but applies *FN* to *ARGS*.

(RETEVAL *POS FORM FLG* —) [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*, and then returns from *POS* with that value. If *FLG* is not NIL and *POS* is a stack pointer, then *POS* is released. The definition of RETEVAL is equivalent to (ENVEVAL *FORM POS* (STKNTH -1 *POS*) *FLG* T), except that RETEVAL does not create a stack pointer.

(RETAPPLY *POS FN ARGS FLG* —) [Function]

Similar to RETEVAL except applies *FN* to *ARGS*.

(RETFROM *POS VAL FLG*) [Function]

Return from the frame specified by the stack descriptor *POS*, with the value *VAL*. If *FLG* is not NIL, and *POS* is a stack pointer, then *POS* is released. An attempt to RETFROM the top level (e.g., (RETFROM T)) causes an error, ILLEGAL STACK ARG. RETFROM can be written in terms of ENVEVAL as follows:

```
(RETFROM
  (LAMBDA (POS VAL FLG)
    (ENVEVAL (LIST 'QUOTE VAL)
             NIL
             (if (STKNTH -1 POS (if FLG then POS))
                else (ERRORX (LIST 19 POS)))
            NIL
            T)))
```

(RETTO *POS VAL FLG*) [Function]

Like RETFROM, except returns *to* the frame specified by *POS*.

(EVALV *VAR POS*) [Function]

Evaluates *VAR*, where *VAR* is assumed to be a litatom, in the access environment specifed by the stack descriptor *POS*. If *VAR* is unbound, EVALV returns NOBIND and does not generate an error. While EVALV could be defined as (ENVEVAL *VAR POS*) it is in fact a SUBR which is somewhat faster. EVALV compiles open when *POS*=NIL.

The following functions and variables are used to manipulate stack pointers.

(STACKP *X*) [Function]

Returns *X* if *X* is a stack pointer, otherwise returns NIL.

(RELSTK *POS*) [Function]

Release the stack pointer *POS* (see page 7.10). If *POS* is not a stack pointer, does nothing. Returns *POS*.

(RELSTKP *X*) [Function]

Returns T is *X* is a released stack pointer, NIL otherwise.

(CLEARSTK *FLG*) [Function]

If *FLG* is NIL, releases all active stack pointers, and returns NIL. If *FLG* is T, returns a list of all the active (unreleased) stack pointers.

CLEARSTKLST [Variable]

A variable used by top-level EVALQT. Every time EVALQT is re-entered (e.g., following errors, or control-D), CLEARSTKLST is checked. If its value is T, all active stack pointers are released using CLEARSTK. If its value is a list, then all stack pointers on that list are released. If its value is NIL, nothing is released. CLEARSTKLST is initially T.

NOCLEARSTKLST [Variable]

A variable used by top-level EVALQT. If CLEARSTKLST is T (see above) all active stack pointers *except* those on NOCLEARSTKLST are released. NOCLEARSTKLST is initially NIL.

Thus if one wishes to use multiple environments that survive through control-D, either CLEARSTKLST should be set to NIL, or else those stack pointers to be retained should be explicitly added to NOCLEARSTKLST.

(COPYSTK *POS1 POS2*) [Function]

(Interlisp-10) Copies the stack, including basic frames, from the frame specified by the stack descriptor *POS1* to the frame specified by the stack descriptor *POS2*. That is, copies the frame extensions and basic frames in the access chain from *POS2* to *POS1* (inclusive). *POS1* must be in the access chain of *POS2*, i.e., "above" *POS2*. Returns the new *POS2*. This provides a way to save an entire environment including variable bindings.

(MAPDL *MAPDLFN MAPDLPOS*) [Function]

Starts at *MAPDLPOS* and applies *MAPDLFN*, a function of two arguments, to the function *name* at each frame, and the frame (stack pointer) itself, until the top of the stack is reached. Returns NIL. For example,

```
[MAPDL (FUNCTION (LAMBDA (X POS)
                   (if (IGREATERP (STKNARGS POS) 2)
                    then (PRINT X)]
```

will print all functions of more than two arguments.

(SEARCHPDL *SRCHFN SRCHPOS*) [Function]

Similar to MAPDL, except searches the pushdown list starting at position *SRCHPOS* until it finds a frame for which *SRCHFN*, a function of two arguments applied to the *name* of the frame and the frame itself, is not NIL. Returns (*NAME* . *FRAME*) if such a frame is found, otherwise NIL.

(BACKTRACE *IPOS EPOS FLAGS FILE PRINTFN*) [Function]

Performs a backtrace beginning at the frame specified by the stack descriptor *IPOS*, and ending with the frame specified by the stack descriptor *EPOS*. *FLAGS* is a number in which the options of the BACKTRACE are encoded. If a bit is set, the corresponding information is included in the backtrace.

bit 0 - print arguments of non-SUBRs.

bit 1 - print temporaries of the interpreter.

bit 2 - print SUBR arguments and local variables.

bit 3 - omit printing of UNTRACE: and function names.

bit 4 - follow access chain instead of control chain.

bit 5 - print temporaries, i.e. the blips.

For example: if *FLAGS*=47Q, everything is printed; if *FLAGS*=21Q, follows the access chain, prints arguments.

FILE is the file that the backtrace is printed to. *FILE* must be open. *PRINTFN* is used when printing the values of variables, temporaries, blips, etc. *PRINTFN*=NIL defaults to PRINT.

(BAKTRACE *IPOS EPOS SKIPFNS FLAGS FILE*) [Function]

Prints a backtrace from *IPOS* to *EPOS* onto *FILE*. *FLAGS* specifies the options of the backtrace, e.g., do/don't print arguments, do/don't print temporaries of the interpreter, etc., and is the same as for BACKTRACE.[2]

[2] BAKTRACE calls BACKTRACE with a *PRINTFN* of SHOWPRINT (page 6.17), so that if SYSPRETTYFLG=T, the values will be prettyprinted.

SKIPFNS is a list of functions. As BAKTRACE scans down the stack, the stack name of each frame is passed to each function in *SKIPFNS*, and if any of them return non-NIL, *POS* is skipped (including all variables).

BAKTRACE collapses the sequence of several function calls corresponding to a call to a system package into a single "function" using BAKTRACELST as described below. For example, any call to the editor is printed as **EDITOR**, a break is printed as **BREAK**, etc.

BAKTRACE is used by the BT, BTV, BTV+, BTV*, and BTV! commands, with *FLAGS*=0, 1, 5, 7, and 47Q respectively.

BAKTRACELST [Variable]

Used for telling BAKTRACE (therefore, the BT, BTV, etc. commands) to abbreviate various sequences of function calls on the stack by a single key, e.g. **BREAK**, **EDITOR**, etc.

The operation of BAKTRACE and format of BAKTRACELST is described so that the user can add his own entries to BAKTRACELST. Each entry on BAKTRACELST is a list of the form (*FRAMENAME KEY* . *PATTERN*) or (*FRAMENAME* (KEY_1 . $PATTERN_1$) ··· (KEY_N . $PATTERN_N$)), where a pattern is a list of elements that are either atoms, which match a single frame, or lists, which are interpreted as a list of alternative patterns, e.g. (PROGN **BREAK** EVAL ((ERRORSET BREAK1A BREAK1) (BREAK1)))

BAKTRACE operates by scanning up the stack and, at each point, comparing the current frame name, with the frame names on BAKTRACELST, i.e. it does an ASSOC. If the frame name does appear, BAKTRACE attempts to match the stack as of that point with (one of) the patterns. If the match is successful, BAKTRACE prints the corresponding key, and continues with where the match left off. If the frame name does not appear, or the match fails, BAKTRACE simply prints the frame name and continues with the next higher frame (unless the *SKIPFNS* applied to the frame name are non-NIL as described above).

Matching is performed by comparing atoms in the pattern with the current frame name, and matching lists as patterns, i.e. sequences of function calls, always working up the stack. For example, either of the sequence of function calls "··· BREAK1 BREAK1A ERRORSET EVAL PROGN ···" or "··· BREAK1 EVAL PROGN ···" would match with the sample entry given above, causing **BREAK** to be printed.

Special features:

- The litatom & can be used to match any frame.

- The pattern "-" can be used to match nothing. - is useful for specifying an optional match, e.g. the example above could also have been written as (PROGN **BREAK** EVAL ((ERRORSET BREAK1A) -) BREAK1).

- It is not necessary to provide in the pattern for matching dummy frames, i.e. frames for which DUMMYFRAMEP (see page 7.4) is true, e.g. in Interlisp-10, *PROG*LAM, *ENV*, NOLINKDEF1, etc. When working on a match, the matcher automatically skips over these frames when they do not match.

- If a match succeeds and the *KEY* is NIL, nothing is printed. For example, (*PROG*LAM NIL EVALA *ENV). This sequence will occur following an error which then causes a break if some of the function's arguments are LOCALVARS.

7.3 RELEASING AND REUSING STACK POINTERS

The creation of a single stack pointer can result in the retention of a large amount of stack space. Furthermore, this space will not be freed until the next garbage collection, *even if the stack pointer is no longer being used*, unless the stack pointer is explicitly released or reused. If there is sufficient amount of stack space tied up in this fashion, a STACK OVERFLOW condition can occur, even in the simplest of computations. For this reason, the user should consider releasing a stack pointer when the environment referenced by the stack pointer is no longer needed.

The effects of releasing a stack pointer are:

(1) The link between the stack pointer and the stack is broken by setting the contents of the stack pointer to the "released mark" (currently unboxed 0). A released stack pointer prints as ***#ADR/#0***.

(2) If this stack pointer was the last remaining reference to a frame extension; that is, if no other stack pointer references the frame extension and the extension is not contained in the active control or access chain, then the extension may be reclaimed, and is reclaimed immediately. The process repeats for the access and control chains of the reclaimed extension so that all stack space that was reachable only from the released stack pointer is reclaimed.

A stack pointer may be released using the function RELSTK, but there are some cases for which RELSTK is not sufficient. For example, if a function contains a call to RETFROM in which a stack pointer was used to specify where to return to, it would not be possible to simultaneously release the stack pointer. (A RELSTK appearing in the function following the call to RETFROM would not be executed!) To permit release of a stack pointer in this situation, the stack functions that relinquish control have optional flag arguments to denote whether or not a stack pointer is to be released (*AFLG* and *CFLG*). Note that in this case releasing the stack pointer will *not* cause the stack space to be reclaimed immediately because the frame referenced by the stack pointer will have become part of the active environment.

Another way of avoiding creating new stack pointers is to *reuse* stack pointers that are no longer needed. The stack functions that create stack pointers (STKPOS, STKNTH, and STKSCAN) have an optional argument which is a stack pointer to reuse. When a stack pointer is reused, two things happen. First the stack pointer is released (see above). Then the pointer to the new frame extension is deposited in the stack pointer. The old stack pointer (with its new contents) is the value of the function. Note that the reused stack pointer will be released even if the function does not find the specified frame.

Note that even if stack pointers are explicitly being released, *creation* of many stack pointers can cause a garbage collection of stack pointer space. Thus, if the user's application requires creating many stack pointers, he definitely should take advantage of reusing stack pointers.

7.4 THE PUSH-DOWN LIST AND THE INTERPRETER

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the following definition of the function

FACT (intentionally faulty):

```
(FACT
   [LAMBDA (N)
      (COND
         ((ZEROP N)
           L)
         (T (ITIMES N (FACT (SUB1 N])
```

In evaluating the form (FACT 1), as soon as FACT is entered, the interpreter begins evaluating the implicit PROGN following the LAMBDA. The first function entered in this process is COND. COND begins to process its list of clauses. After calling ZEROP and getting a NIL value, COND proceeds to the next clause and evaluates T. Since T is true, the evaluation of the implicit PROGN that is the consequent of the T clause is begun. This requires calling the function ITIMES. However before ITIMES can be called, its arguments must be evaluated. The first argument is evaluated by retrieving the current binding of N from its value cell; the second involves a recursive call to FACT, and another implicit PROGN, etc.

Note that at each stage of this process, some portion of an expression has been evaluated, and another is awaiting evaluation. The output below (from Interlisp-10) illustrates this by showing the state of the push-down list at the point in the computation of (FACT 1) when the unbound atom L is reached.

```
←FACT(1)
u.b.a. L {in FACT} in ((ZEROP N) L)
(L broken)
:BTV!

   *TAIL* (L)

   *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
COND

   *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
   *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))

   N 0
FACT


   *FORM* (FACT (SUB1 N))
   *FN* ITIMES
   *TAIL* ((FACT (SUB1 N)))
   *ARGVAL* 1
   *FORM* (ITIMES N (FACT (SUB1 N)))
   *TAIL* ((ITIMES N (FACT (SUB1 N))))

   *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
COND

   *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
   *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
   N 1
FACT

**TOP**
```

Internal calls to EVAL, e.g., from COND and the interpreter, are marked on the push-down list by a special mark or blip which the backtrace prints as *FORM*. The genealogy of *FORM*'s is thus a history of the computation. Other temporary information stored on the stack by the interpreter includes the tail of a partially evaluated implicit PROGN (e.g., a cond clause or lambda expression) and the tail of a partially evaluated form (i.e., those arguments not yet evaluated), both indicated on the backtrace by *TAIL*, the values of arguments that have already been evaluated, indicated by *ARGVAL*, and the names of functions waiting to be called, indicated by *FN*. *ARG1, ···, *ARGn are used by the backtrace to indicate the (unnamed) arguments to SUBRs.

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated (except for nlambda functions, of course). Also note that the *ARG1, *FORM*, *TAIL*, etc. "bindings" comprise the actual working storage. In other words, in the above example, if a (lower) function changed the value of the *ARG1 binding, the COND would continue interpreting the new binding as a list of COND clauses. Similarly, if the *ARGVAL* binding were changed, the new value would be given to ITIMES as its first argument after its second argument had been evaluated, and ITIMES was actually called.

Note that *FORM*, *TAIL*, *ARGVAL*, etc., do not actually appear as variables on the stack, i.e., evaluating *FORM* or calling STKSCAN to search for it will not work. However, the functions BLIPVAL, SETBLIPVAL, and BLIPSCAN described below are available for accessing these internal blips. These functions currently know about four different types of blips:

FN	the name of a function about to be called
ARGVAL	an argument for a function about to be called
FORM	a form in the process of evaluation
TAIL	the tail of a COND clause, implicit PROGN, PROG, etc.

(BLIPVAL *BLIPTYP IPOS FLG*) [Function]

Returns the value of the specified blip of type *BLIPTYP*. If *FLG* is a number N, finds the Nth blip of the desired type, searching the control chain beginning at the frame specified by the stack descriptor *IPOS*. If *FLG* is NIL, 1 is used. If *FLG* is T, returns the number of blips of the specified type at *IPOS*.

(SETBLIPVAL *BLIPTYP IPOS N VAL*) [Function]

Sets the value of the specified blip of type *BLIPTYP*. Searches for the *N*th blip of the desired type, beginning with the frame specified by the stack descriptor *IPOS*, and following the control chain.

(BLIPSCAN *BLIPTYP IPOS*) [Function]

Returns a stack pointer to the frame in which a blip of type *BLIPTYP* is located. Search begins at the frame specified by the stack descriptor *IPOS* and follows the control chain.

7.5 GENERATORS AND COROUTINES

This section describes an application of the spaghetti stack facility to provide mechanisms for creating and using simple generators, generalized coroutines, and Conniver style possibility lists.

7.5.1 Generators

A *generator* is like a subroutine except that it retains information about previous times it has been called. Some of this state may be data (for example, the seed in a random number generator), and some may be in program state (as in a recursive generator which finds all the atoms in a list structure). For example, if LISTGEN is defined as:

```
(LISTGEN (L)
      (IF L THEN (PRODUCE (CAR L))
                 (LISTGEN (CDR L))))
```

we can use the function GENERATOR (described below) to create a generator that uses LISTGEN to produce the elements of a list one at a time, e.g.,

```
(SETQ GR (GENERATOR (LISTGEN '(A B C)))
```

creates a generator, which can be called by

```
(GENERATE GR)
```

to produce as values on successive calls, A, B, C. When GENERATE (not GENERATOR) is called the first time, it simply starts evaluating (LISTGEN '(A B C)). PRODUCE gets called from LISTGEN, and pops back up to GENERATE with the indicated value after saving the state. When GENERATE gets called again, it continues from where the last PRODUCE left off. This process continues until finally LISTGEN completes and returns a value (it doesn't matter what it is). GENERATE then returns GR itself as its value, so that the program that called GENERATE can tell that it is finished, i.e., there are no more values to be generated.

(GENERATOR *FORM##* *COMVAR##*) [NLambda Function]

An nlambda function that creates a generator which uses *FORM##* to compute values. GENERATOR returns a *generator handle* which is represented by a dotted pair of stack pointers.

COMVAR## is optional. If its value (EVAL of) is a generator handle, the list structure and stack pointers will be reused. Otherwise, a new generator handle will be constructed.

GENERATOR compiles open.

(PRODUCE *VAL*) [Function]

Used from within (below) a generator to return *VAL* as the value of the corresponding call to GENERATE.

(GENERATE *HANDLE* *VAL*) [Function]

Restarts the generator represented by *HANDLE*. *VAL* is returned as the value of the PRODUCE which last suspended the operation of the generator. When the generator runs out of values, GENERATE returns *HANDLE* itself.

Examples:

The following function will go down recursively through a list structure and produce the atoms in the list structure one at a time.

```
[LEAVESG (L)
    (if (ATOM L)
     then (PRODUCE L)
     else (LEAVESG (CAR L))
          (if (CDR L)
            then (LEAVESG (CDR L)]
```

The following function prints each of these atoms as it appears. It illustrates how a loop can be set up to use a generator.

```
(PLEAVESG1 (L)
   (PROG (X LHANDLE)
         (SETQ LHANDLE (GENERATOR (LEAVESG L)))
     LP  (SETQ X (GENERATE LHANDLE))
         (if (EQ X LHANDLE)
             then (RETURN NIL))
         (PRINT X)
         (GO LP)))
```

Note that the loop terminates when the value of the generator is EQ to the dotted pair which is the value produced by the call to GENERATOR. A CLISP iterative operator, OUTOF, is provided which makes it much easier to write the loop in PLEAVESG1. OUTOF (or outof) can precede a form which is to be used as a generator. On each iteration, the iteration variable will be set to successive values returned by the generator; the loop will be terminated automatically when the generator runs out. Therefore, the following is equivalent to the above program PLEAVESG1:

```
(PLEAVESG2 (L)
   (for X outof (LEAVESG L) do (PRINT x))
```

Here is another example; the following form will print the first N atoms.

```
(for X outof (MAPATOMS (FUNCTION PRODUCE))
   as I from 1 to N do (PRINT X))
```

7.5.2 Coroutines

This package provides facilities for the creation and use of fully general coroutine structures. It uses a stack pointer to preserve the state of a coroutine, and allows arbitrary switching between *N* different coroutines, rather than just a call to a generator and return. This package is slightly more efficient than the generator package described above, and allows more flexibility on specification of what to do when a coroutine terminates.

(COROUTINE *CALLPTR## COROUTPTR## COROUTFORM## ENDFORM##*) [NLambda Function]

This nlambda function is used to create a coroutine and initialize the linkage. *CALLPTR##* and *COROUTPTR##* are the names of two variables, which will be set to appropriate stack pointers. If the values of *CALLPTR##* or *COROUTPTR##* are already stack pointers, the stack pointers will be reused. *COROUTFORM##* is the form which is evaluated to start the coroutine; *ENDFORM##* is a form to be evaluated if *COROUTFORM##* actually returns when it runs out of values.

COROUTINE compiles open.

(RESUME *FROMPTR TOPTR VAL*) [Function]

Used to transfer control from one coroutine to another. *FROMPTR* should be the stack pointer for the current coroutine, which will be smashed to preserve the current state. *TOPTR* should be the stack pointer which has preserved the state of the coroutine to be transferred to, and *VAL* is the value that is to be returned to the latter coroutine as the value of the RESUME which suspended the operation of that coroutine.

For example, the following is the way one might write the LEAVES program using the coroutine package:

```
(LEAVESC (L COROUTPTR CALLPTR)
   (if (ATOM L)
    then (RESUME COROUTPTR CALLPTR L)
    else (LEAVESC (CAR L) COROUTPTR CALLPTR)
         (if (CDR L) then (LEAVESC (CDR L) COROUTPTR CALLPTR))))
```

A function PLEAVESC which uses LEAVESC can be defined as follows:

```
(PLEAVESC (L)
  (bind PLHANDLE LHANDLE
   first (COROUTINE PLHANDLE LHANDLE
                    (LEAVESC L LHANDLE PLHANDLE)
                    (RETFROM 'PLEAVESC))
   do (PRINT (RESUME PLHANDLE LHANDLE))))
```

By RESUMEing LEAVESC repeatedly, this function will print all the leaves of list L and then return out of PLEAVESC via the RETFROM. The RETFROM is necessary to break out of the non-terminating do-loop. This was done to illustrate the additional flexibility allowed through the use of *ENDFORM##*.

We use two coroutines working on two trees in the example EQLEAVES, defined below. EQLEAVES tests to see whether two trees have the same leaf set in the same order, e.g., (EQLEAVES '(A B C) '(A B (C))) is true.

```
(EQLEAVES (L1 L2)
    (bind LHANDLE1 LHANDLE2 PE EL1 EL2
     first (COROUTINE PE LHANDLE1 (LEAVESC L1 LHANDLE1 PE) 'NO-MORE)
           (COROUTINE PE LHANDLE2 (LEAVESC L2 LHANDLE2 PE) 'NO-MORE)
     do (SETQ EL1 (RESUME PE LHANDLE1))
        (SETQ EL2 (RESUME PE LHANDLE2))
        (if (NEQ EL1 EL2)
         then (RETURN NIL))
    repeatuntil (EQ EL1 'NO-MORE)
    finally (RETURN T)))
```

7.5.3 Possibilities Lists

A possibilities list is the interface between a generator and a consumer. The possibilities list is initialized by a call to POSSIBILITIES, and elements are obtained from it by using TRYNEXT. By using the spaghetti stack to maintain separate environments, this package allows a regime in which a generator can put a few items in a possibilities list, suspend itself until they have been consumed, and be subsequently aroused and generate some more.

(POSSIBILITIES *FORM##*) [NLambda Function]

This nlambda function is used for the initial creation of a possibilities list. *FORM##* will be evaluated to create the list. It should use the functions NOTE and AU-REVOIR described below to generate possibilities. Normally, one would set some variable to the possibilities list which is returned, so it can be used later, e.g.:

(SETQ PLIST (POSSIBILITIES (GENERFN V1 V2))).

POSSIBILITIES compiles open.

(NOTE *VAL LSTFLG*) [Function]

Used within a generator to put items on the possibilities list being generated. If *LSTFLG* is equal to NIL, *VAL* is treated as a single item. If *LSTFLG* is non-NIL, then the list *VAL* is NCONCed on the end of the possibilities list. Note that it is perfectly reasonable to create a possibilities list using a second generator, and NOTE that list as possibilities for the current generator with *LSTFLG* equal to T. The lower generator will be resumed at the appropriate point.

(AU-REVOIR *VAL##*) [NoSpread Function]

Puts *VAL##* on the possibilities list if it is given, and then suspends the generator and returns to the consumer in such a fashion that control will return to the generator at the AU-REVOIR if the consumer exhausts the possibilities list.

Note: NIL is not put on the possibilities list unless it is explicitly given as an argument to AU-REVOIR, i.e., (AU-REVOIR) and (AU-REVOIR NIL) are *not* the same. AU-REVOIR and ADIEU are lambda nospreads to enable them to distinguish these two cases.

(ADIEU *VAL##*) [NoSpread Function]

Like AU-REVOIR except releases the generator instead of suspending it.

(TRYNEXT *PLST## ENDFORM## VAL##*) [NLambda Function]

This nlambda function allows a consumer to use a possibilities list. It removes the first item from the possibilities list named by *PLST##* (i.e. *PLST##* must be an atom whose value is a possiblities list), and returns that item, provided it is not a generator handle. If a generator handle is encountered, the generator is reawakened. When it returns a possibilities list, this list is added to the front of the current list. When a call to TRYNEXT causes a generator to be awakened, *VAL##* is returned as the value of the AU-REVOIR which put that generator to sleep. If *PLST##* is empty, it evaluates *ENDFORM##* in the caller's environment.

TRYNEXT compiles open.

(CLEANPOSLST *PLST*) [Function]

This function is provided to release any stack pointers which may be left in the *PLST* which was not used to exhaustion.

For example, FIB is a generator for fibonnaci numbers. It starts out by NOTEing its two arguments, then suspends itself. Thereafter, on being re-awakened, it will NOTE two more terms in the series and suspends again. PRINTFIB uses FIB to print the first N fibonacci numbers.

```
[FIB (F1 F2)
  (do (NOTE F1)
      (NOTE F2)
      (SETQ F1 (IPLUS F1 F2))
      (SETQ F2 (IPLUS F1 F2))
      (AU-REVOIR)]
```

Note that this AU-REVOIR just suspends the generator and adds nothing to the possibilities list except the generator.

```
[PRINTFIB (N)
     (PROG ((FL (POSSIBILITIES (FIB 0 1))))
           (RPTQ N (PRINT  (TRYNEXT FL)))
           (CLEANPOSLST FL)]
```

Note that FIB itself will never terminate.

Possibilities Lists

CHAPTER 8

THE PROGRAMMER'S ASSISTANT

8.1 INTRODUCTION

With any interactive computer language, the user interacts with the system through an "executive", which interprets and executes typed-in commands. In most implementations of Lisp, the executive is a simple "read-eval-print" loop, which repeatedly reads a Lisp expression, evaluates it, and prints out the value of the expression. Interlisp has an executive which allows a much greater range of inputs, other than just regular Interlisp expressions.

In particular, the Interlisp executive implements a facility known as the "programmer's assistant" (or "p.a."). The central idea of the programmer's assistant is that the user is addressing an active intermediary, namely his assistant. Normally, the assistant is invisible to the user, and simply carries out the user's requests. However, the assistant remembers what the user has done, so the user can give commands to repeat a particular operation or sequence of operations, with possible modifications, or to undo the effect of specified operations. Like DWIM, the programmer's assistant embodies an approach to system design whose ultimate goal is to construct an environment that "cooperates" with the user in the development of his programs, and frees him to concentrate more fully on the conceptual difficulties and creative aspects of the problem at hand.

We will first discuss the various input formats, then the use of commands to the programmer's assistant, and finally how to modify the programmer's assistant for specialized uses.

8.1.1 Input Formats

The Interlisp executive accepts inputs in the following formats:

(1) A single litatom, followed by a carriage-return. The value of the litatom is returned. For the purposes of this discussion, we will call this EVALV-format.

(2) A regular Interlisp expression, beginning with a left parenthesis or square bracket and terminated by a matching right parenthesis or square bracket. A right bracket matches any number of left parentheses, back to the last left bracket or the entire expression. Such an input is known as an "EVAL-format" input, since the form is simply passed to EVAL for evaluation. Notice that it is not necessary to type a carriage return at the end of such a form; Interlisp will supply one automatically. If a carriage-return is typed before the final matching right parenthesis or bracket, it is treated as a space, and input continues. The following examples are all interpreted the same:

```
←(PLUS 1 (TIMES 2 3))
```

```
←(PLUS 1 (TIMES 2 3]
```

←(PLUS 1 (TIMES^{cr}
2 3]

(3) Often, the user, typing at the keyboard, calls functions with constant argument values, which would have to be quoted if the user typed it in "EVAL-format". For convience, if the user types a litatom immediately followed by a list form, the litatom is APPLYed to the elements within the list, unevaluated. For example, typing LOAD(FOO) is equivalent to typing (LOAD 'FOO), and GETPROP(X COLOR) is equivalent to (GETPROP 'X 'COLOR). The input is terminated by the matching right parenthesis or bracket. We will call such input "APPLY-format." APPLY-format input is useful in some situations, but note that it may produce unexpected results when an *nlambda* function is called that explicitly evaluates its arguments. For example, typing SETQ(FOO BAR) will set FOO to the *value* of BAR, not to BAR itself.

However, there are times when a user does not want to terminate the input when a closing parenthesis is typed — especially when giving a command to the programmer's assistant. This leads us to our fourth format.

(4) A sequence of litatoms and lists *beginning with* a litatom and a space (to distinguish it from APPLY-format), terminated by a carriage return or an extra right parenthesis or bracket. If a list is terminated then Interlisp will type a carriage-return and "..." to indicate that further input will be accepted. The user can type further expressions or terminate the whole expression by a carriage-return.

Once the input is terminated, the programmer's assistant decides how to evaluate the expression. This determination relies on a heuristic that says "If there is only expression, then assume EVALV-format. If there are two expressions, then assume APPLY-format. If there are three or more expressions, then assume EVAL-format." The following inputs are examples of this rule:

←FOO<space>^{cr}

same as FOO^{cr} — EVALV-format

←LIST (A B)
...^{cr}

same as LIST(A B) — APPLY-format

←PLUS (TIMES 2 3)
...1^{cr}

same as (PLUS (TIMES 2 3) 1) — EVAL-format

8.1.2 Examples

So far, we have dealt only with how the executive instructs Interlisp to evaluate input. However, the same scheme also allows the user to give commands directly to the programmer's assistant. In fact, in each of the above cases, it is first determined whether the initial litatom is a command to the programmer's assistant. If so, the normal lisp evaluation process is bypassed. Note that this means that a function or variable with the same name as a programmer's assistant command will not be evaluated (in the normal lisp sense) if it is the first litatom of an expression input to the executive.

The programmer's assistant facility features the use of memory structures called "history lists." A history list is a list of the information associated with each of the individual "events" that have occurred in the system, where each event corresponds to one user input. Associated with each event on the history list is the input and its value, plus other optional information such as side-effects, formatting information, etc.

The following dialogue, taken from an actual session at the terminal, contains illustrative (but not necessarily useful) examples and gives the flavor of the programmer's assistant facility in Interlisp. The number before each prompt is the "event number" (see page 8.26).

```
12←(SETQ FOO 5)
5
13←(SETQ FOO 10)
(FOO reset)
10
```

The p.a. notices that the user has reset the value of FOO and informs the user.

```
14←UNDO
SETQ undone.
15←FOOcr
5
```

This is the first example of direct communication with the p.a. The user has said to UNDO the previous input to the executive.

.
.
.

```
25←SET(LST1 (A B C))
(A B C)
26←(SETQ LST2 '(D E F))
(D E F)
27←(FOR X IN LST1 DO (REMPROP X 'MYPROP]
NIL
```

The user asked to remove the property MYPROP from the atoms A, B, and C. Now lets assume that is not what he wanted to do, but rather use the elements of LST2.

```
28←UNDO FOR
FOR undone.
```

First he undoes the REMPROP, by undoing the iterative statement. Notice the UNDO accepted an "argument," although in this case UNDO by itself would be sufficient.

```
29←USE LST2 FOR LST1 IN 27
NIL
```

The user just instructed to go back to event number 27 and substitute LST2 for LST1 and then reexecute the expression. The user could have also specified -2 instead of 27 to specify a relative address.

.
.
.

```
47←(PUTHASH 'FOO (MKSTRING 'FOO) MYHASHARRAY)
"FOO"
```

If MKSTRING was a computationally expensive function (which it is not), then the user might be cacheing its value for later use.

```
48←USE FIE FUM FOE FOR FOO IN MKSTRING
"FIE"
"FUM"
"FOE"
```

The user now decides he would like to redo the PUTHASH several times with different values. He specifies the event by "IN MKSTRING" rather than PUTHASH.

```
49←?? USE

48.      USE FIE FUM FOE FOR FOO IN MKSTRING
     ←(PUTHASH (QUOTE FIE) (MKSTRING (QUOTE FIE)) MYHASHARRAY)
     "FIE"
     ←(PUTHASH (QUOTE FUM) (MKSTRING (QUOTE FUM)) MYHASHARRAY)
     "FUM"
     ←(PUTHASH (QUOTE FOE) (MKSTRING (QUOTE FOE)) MYHASHARRAY)
     "FOE"
```

Here we see the user ask the p.a. (using the ?? command) what it has on its history list for the last input to the executive. Since the event corresponds to a programmer's assistant command that evaluates several forms, these forms are saved as the input, although the user's actual input, the p.a. command, is also saved in order to clarify the printout of that event.

As stated earlier, the most common interaction with the programmer's assistant occurs at the top level read-eval-print loop, or in a break, where the user types in expressions for evaluation, and sees the values printed out. In this mode, the assistant acts much like a standard Lisp executive, except that before attempting to evaluate an input, the assistant first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or reexecution. The assistant also notes new functions and variables to be added to its spelling lists to enable future corrections. Then the assistant executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the entry on the history list corresponding to the input, and prints the result, followed by a prompt character to indicate it is again ready for input.

If the input typed by the user is recognized as a p.a. command, the assistant takes special action. Commands such as `UNDO` and `??` are immediately performed. Commands that involved reexecution of previous inputs, such as `REDO` and `USE`, are achieved by computing the corresponding input expression(s) and then *unreading* them. The effect of this unreading operation is to cause the assistant's input routine, `LISPXREAD`, to act exactly as though these expressions were typed in by the user. These expressions are processed exactly as though they had been typed, except that they are not saved on new and separate entries on the history list, but associated with the history command that generated them.

The net effect of this implementation of the programmer's assistant is to provide a facility which is easily inserted at many levels, and embodies a consistent set of commands and conventions for talking about past events. This gives the user the subjective feeling that a single agent is watching everything he does and says, and is always available to help.

8.2 PROGRAMMER'S ASSISTANT COMMANDS

The programmer's assistant recognizes a number of commands, which usually refer to past events on the history list. These commands are treated specially; for example, they may not be put on the history list.

Note: If the user defines a function by the same name as a p.a. command, a warning message is printed to remind him that the p.a. command interpretation will take precedence for type-in.

All programmer's assistant commands use the same conventions and syntax for indicating which event or events on the history list the command refers to, even though different commands may be concerned with different aspects of the corresponding event(s), e.g., side-effects, value, input, etc. Therefore, before discussing the various p.a. commands, the following section describes the types of event specifications currently implemented.

8.2.1 Event Specification

An event address identifies one event on the history list. It consists of a sequence of "commands" for moving an imaginary cursor up or down the history list, much in the manner of the arguments to the @ break command (see page 9.3). The event identified is the one "under" the imaginary cursor when there are no more commands. (If any command fails, an error is generated and the history command is aborted.) For example, the event address `42` refers to the event with event number 42, `42 FOO` refers to the first event (searching back from event 42) whose input contains the word `FOO`, and `42 FOO -1` refers to the event preceeding that event. Usually, an event address will contain only one or two commands.

Most of the event address commands perform searches for events which satisfy some condition. Unless the ← command is given (see below), this search always goes backwards through the history list, from the most recent event specified to the oldest. Note that each search skips the current event. For example, if `FOO` refers to event *N*, `FOO FIE` will refer to some event before event *N*, even if there is a `FIE` in event *N*.

The event address commands are interpreted as follows:

N (an integer)

If *N* is the first command in an event address, refers to the event with event number *N*. Otherwise, refers to the event *N* events forward (in direction of increasing event number). If *N* is negative, it always refers to the event -*N* events backwards.

For example, `-1` refers to the previous event, `42` refers to event number 42 (if the first command in an event address), and `42 3` refers to the event with event number 45.

←*LITATOM*

Specifies the last event with an `APPLY`-format input whose *function* matches *LITATOM*.

Note: There must not be a space between ← and *LITATOM*.

←

Specifies that the next search is to go forward instead of backward. If given as the first event address command, the next search begins with last (oldest) event on the history list.

`F`

Specifies that the next object in the event address is to be searched for, regardless of what it is. For example, `F -2` looks for an event containing `-2`.

`=` — Specifies that the next object (presumably a pattern) is to be matched against the *values* of events, instead of the inputs.

`\` — Specifies the event last located.

`SUCHTHAT` *PRED* — Specifies an event for which the function *PRED* returns true. *PRED* should be a function of two arguments, the input portion of the event, and the event itself. See page 8.25 for a discussion of the format of events on the history list.

PAT — Any other event address command specifies an event whose input contains an expression that matches *PAT* as described in page 17.13.

The matching is performed by the function `HISTORYMATCH` (page 8.33), which is initially defined to call `EDITFINDP` but can be advised or redefined for specialized applications.

Note: Symbols used below of the form $EventAddress_i$ refer to event addresses, described above. Since an event address may contain multiple words, the event address is parsed by searching for the words which delimit it. For example, in `FROM` $EventAddress_1$ `THRU` $EventAddress_2$, the symbol $EventAddress_1$ corresponds to all words between `FROM` and `THRU` in the event specification, and $EventAddress_2$ to all words from `THRU` to the end of the event specification.

`FROM` $EventAddress_1$ `THRU` $EventAddress_2$
$EventAddress_1$ `THRU` $EventAddress_2$

Specifies the sequence of events from the event with address $EventAddress_1$ through the event with address $EventAddress_2$. For example, `FROM 47 THRU 49` specifies events 47, 48, and 49. $EventAddress_1$ can be more recent than $EventAddress_2$. For example, `FROM 49 THRU 47` specifies events 49, 48, and 47 (note reversal of order).

`FROM` $EventAddress_1$ `TO` $EventAddress_2$
$EventAddress_1$ `TO` $EventAddress_2$

Same as `THRU` but does not include event $EventAddress_2$.

`FROM` $EventAddress_1$ — Same as `FROM` $EventAddress_1$ `THRU -1`. For example, if the current event is number 53, then `FROM 49` specifies events 49, 50, 51, and 52.

`THRU` $EventAddress_2$ — Same as `FROM -1 THRU` $EventAddress_2$. For example, if the current event is number 53, then `THRU 49` specifies events 52, 51, 50, and 49 (note reversal of order).

`TO` $EventAddress_2$ — Same as `FROM -1 TO` $EventAddress_2$.

`ALL` $EventAddress_1$ — Specifies all events satisfying $EventAddress_1$. For example, `ALL LOAD`, `ALL SUCHTHAT FOO`.

empty — If nothing is specified, it is the same as specifying `-1`.

Note: In the special case that the last event was an `UNDO`, it is the same as specifying `-2`. For example, if the user types `(NCONC FOO FIE)`, he can then type `UNDO`, followed by `USE NCONC1`.

EventSpec$_1$ AND *EventSpec*$_2$ AND ··· AND *EventSpec*$_N$

Each of the *EventSpec*$_i$ is an event specification. The lists of events are concatenated. For example, FROM 30 THRU 32 AND 35 THRU 37 is the same as 30 AND 31 AND 32 AND 35 AND 36 AND 37.

@ *LITATOM*

If *LITATOM* is the name of a command defined via the NAME command (page 8.12), specifies the event(s) defining *LITATOM*.

@@ *EventSpec*

EventSpec is an event specification interpreted as above, but with respect to the archived history list (see page 8.13).

If no events can be found that satisfy the event specification, spelling correction on each word in the event specification is performed using LISPXFINDSPLST as the spelling list. For example, REDO 3 THRUU 6 will work correctly. If the event specification still fails to specify any events after spelling correction, an error is generated.

8.2.2 Commands

All programmer's assistant commands can be input as list forms, or as lines (see page 8.30). For example, typing REDO 5^{cr} and (REDO 5) are equivalent.

EventSpec is used to denote an event specification. Unless specified otherwise, omitting *EventSpec* is the same as specifying *EventSpec*=-1. For example, REDO and REDO -1 are the same.

REDO *EventSpec* [Prog. Asst. Command]

Redoes the event or events specified by *EventSpec*. For example, REDO FROM -3 redoes the last three events.

REDO *EventSpec* *N* TIMES [Prog. Asst. Command]

Redoes the event or events specified by *EventSpec* *N* times. For example, REDO 10 TIMES redoes the last event ten times.

REDO *EventSpec* WHILE *FORM* [Prog. Asst. Command]

Redoes the specified events as long as the value of *FORM* is true. *FORM* is evaluated before each iteration so if its initial value is NIL, nothing will happen.

REDO *EventSpec* UNTIL *FORM* [Prog. Asst. Command]

Same as REDO *EventSpec* WHILE (NOT *FORM*).

REPEAT *EventSpec* [Prog. Asst. Command]

Same as REDO *EventSpec* WHILE T. The event(s) are repeated until an error occurs, or the user types control-E or control-D.

REPEAT *EventSpec* WHILE *FORM* [Prog. Asst. Command]
REPEAT *EventSpec* UNTIL *FORM* [Prog. Asst. Command]

Same as REDO.

For all history commands that perform multiple repetitions, the variable REDOCNT is initialized to 0 and incremented each iteration. If the event terminates gracefully, i.e., is not aborted by an error or control-D, the number of iterations is printed.

RETRY *EventSpec* [Prog. Asst. Command]

Similar to REDO except sets HELPCLOCK (page 9.11) so that any errors that occur while executing *EventSpec* will cause breaks.

USE *EXPRS* FOR *ARGS* IN *EventSpec* [Prog. Asst. Command]

Substitutes *EXPRS* for *ARGS* in *EventSpec*, and redoes the result. Substitution is done by ESUBST (page 17.57), and is carried out as described below. *EXPRS* and *ARGS* can include non-atomic members.

For example, USE LOG (MINUS X) FOR ANTILOG X IN -2 AND -1 will substitute LOG for every occurrence of ANTILOG in the previous two events, and substitute (MINUS X) for every occurrence of X, and reexecute them. Note that these substitutions do not change the information saved about these events on the history list.

Any expression to be substituted can be preceded by a !, meaning that the expression is to be substituted as a *segment*, e.g., LIST(A B C) followed by USE ! (X Y Z) FOR B will produce LIST(A X Y Z C), and USE ! NIL FOR B will produce LIST(A C).

If IN *EventSpec* is omitted, the first member of *ARGS* is used for *EventSpec*. For example, USE PUTD FOR @UTD is equivalent to USE PUTD FOR @UTD IN F @UTD. The F is inserted to handle correctly the case where the first member of *ARGS* could be interpreted as an event address command.

USE *EXPRS* IN *EventSpec* [Prog. Asst. Command]

If *ARGS* are omitted, and the event referred to was itself a USE command, the arguments and expression substituted into are the same as for the indicated USE command. In effect, this USE command is thus a continuation of the previous USE command. For example, following USE X FOR Y IN 50, typing USE Z IN -1 is equivalent to USE Z FOR Y IN 50.

If *ARGS* are omitted and the event referred to was *not* a USE command, substitution is for the "operator" in that command. For example ARGLIST(FF) followed by USE CALLS IN -1 is equivalent to USE CALLS FOR ARGLIST IN -1.

If IN *EventSpec* is omitted, it is the same as specifying IN -1.

USE $EXPRS_1$ FOR $ARGS_1$ AND ··· AND $EXPRS_N$ FOR $ARGS_N$ IN *EventSpec* [Prog. Asst. Command]

More general form of USE command. See description of the substitution algorithm below.

Note: The USE command is parsed by a small finite state parser to distinguish the expressions and arguments. For example, USE FOR FOR AND AND AND FOR FOR will be parsed correctly.

Every USE command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event specification, which defines the input expression in which the substitution takes place. If the USE command has the same number of expressions as arguments, the substitution procedure is straightforward.[1] For example, USE X Y FOR U V means substitute X for U and Y for V, and is equivalent to USE X FOR U AND Y FOR V. However, the USE command also permits distributive substitutions, for substituting several expressions for the same argument. For example, USE A B C FOR X means first substitute A for X then substitute B for X (in a new copy of the expression), then substitute C for X. The effect is the same as three separate USE commands. Similarly, USE A B C FOR D AND X Y Z FOR W is equivalent to USE A FOR D AND X FOR W, followed by USE B FOR D AND Y FOR W, followed by USE C FOR D AND Z FOR W. USE A B C FOR D AND X FOR Y also corresponds to three substitions, the first with A for D and X for Y, the second with B for D, and X for Y, and the third with C for D, and again X for Y. However, USE A B C FOR D AND X Y FOR Z is ambiguous and will cause an error. Essentially, the USE command operates by proceeding from left to right handling each "AND" separately. Whenever the number of expressions exceeds the number of expressions available, multiple USE expressions are generated. Thus USE A B C D FOR E F means substitute A for E at the same time as substituting B for F, then in another copy of the indicated expression, substitute C for E and D for F. Note that this is also equivalent to USE A C FOR E AND B D FOR F.

... *VARS* [Prog. Asst. Command]

Similar to USE except substitutes for the (first) *operand.*

For example, EXPRP(FOO) followed by ... FIE FUM is equivalent to USE FIE FUM FOR FOO.

Note: In the following discussion, $ is used to represent the character <esc>, since this is how <esc> is echoed.

$ *X* FOR *Y* IN *EventSpec* [Prog. Asst. Command]

$ is a special form of the USE command for conveniently specifying *character* substitutions in litatoms or strings. In addition, it has a number of useful properties in connection with events that involve errors (see below).

Equivalent to USE $*X*$ FOR $*Y*$ IN *EventSpec*, which will do a character substitution of the characters in *X* for the characters in *Y*.

For example, if the user types MOVD(FOO FOOSAVE T), he can then type $ FIE FOR FOO IN MOVD to perform MOVD(FIE FIESAVE T). Note that USE FIE FOR FOO would perform MOVD(FIE FOOSAVE T).

$ *Y* *X* IN *EventSpec* [Prog. Asst. Command]
$ *Y* TO *X* IN *EventSpec* [Prog. Asst. Command]
$ *Y* = *X* IN *EventSpec* [Prog. Asst. Command]
$ *Y* -> *X* IN *EventSpec* [Prog. Asst. Command]

Abbreviated forms of the $ command: the same as $ *X* FOR *Y* IN *EventSpec*, which changes *Y*s to *X*s.

$ does event location the same as the USE command, i.e., if IN *EventSpec* is not specified, $ searches for *Y*. However, unlike USE, $ can only be used to specify one substitution at a time. After $ finds the event, it looks to see if an error was involved in that event, and if the indicated character substitution can be performed in the object of the error message, called the offender. If so, $ assumes the substitution refers

[1] Except when one of the arguments and one of the expressions are the same, e.g., USE X Y FOR Y X, or USE X FOR Y AND Y FOR X. This situation is noticed when parsing the command, and handled correctly.

to the offender, performs the indicated character substitution in the offender only, and then substitutes the result for the original offender throughout the event. For example, suppose the user types (PRETTYDEF FOOFNS 'FOO FOOOVARS) causing a U.B.A. FOOOVARS error message. The user can now type $ OO O, which will change FOOOVARS to FOOVARS, but *not* change FOOFNS or FOO.

If an error did occur in the specified event, the user can also omit specifying the object of the substitution, *Y*, in which case the offender itself is used. Thus, the user could have corrected the above example by simply typing $ FOOVARS. Since ESUBST is used for performing the substitution (see page 17.57), $ can be used in *X* to refer to the characters in *Y*. For example, if the user types LOAD(PRSTRUC PROP), causing the error FILE NOT FOUND PRSTRUC, he can request the file to be loaded from LISP's directory by simply typing $ <LISP>$. This is equivalent to performing (R PRSTRUC <LISP>$) on the event, and therefore replaces PRSTRUC by <LISP>PRSTRUC.

Note that $ never *searches* for an error. Thus, if the user types LOAD(PRSTRUC PROP) causing a FILE NOT FOUND error, types CLOSEALL(), and *then* types $ <LISP>$, LISPX will complain that there is no error in CLOSEALL(). In this case, the user would have to type $ <LISP>$ IN LOAD, or $ PRS <LISP>PRS (which would cause a search for PRS).

Note also that $ operates on *input*, not on programs. If the user types FOO(), and within the call to FOO gets a U.D.F. CONDD error, he *cannot* repair this by $ COND. LISPX will type CONDD NOT FOUND IN FOO().

FIX *EventSpec* [Prog. Asst. Command]

Envokes the default program editor (Dedit or the teletype editor) on a copy of the input(s) for *EventSpec*. Whenever the user exits via OK, the result is unread and reexecuted exactly as with REDO.

FIX is provided for those cases when the modifications to the input(s) are not simple substitutions of the type that can be specified by USE. For example, if the default editor is the teletype editor, then:

```
←(DEFINEQ FOO (LAMBDA (X) (FIXSPELL SPELLINGS2 X 70]
INCORRECT DEFINING FORM
FOO
←FIX
EDIT
*P
(DEFINEQ FOO (LAMBDA & &))
*(LI 2)
*P
(DEFINEQ (FOO &))
*OK
(FOO)
←
```

The user can also specify the edit command(s) to LISPX, by typing - followed by the command(s) after the event specification, e.g., FIX - (LI 2). In this case, the editor will not type EDIT, or wait for an OK after executing the commands.

Note: FIX calls the editor on the "input sequence" of an event, adjusting the editor so it is initially editing the expression typed. However, the entire input sequence is being edited, so it is possible to give editor commands that examine this structure further. For more information on the format of an event's input, see page 8.25.

?? *EventSpec* [Prog. Asst. Command]

Prints the specified events from the history list. If *EventSpec* is omitted, ?? prints the entire history list, beginning with most recent events. Otherwise ?? prints only those events specified in *EventSpec* (in the order specified). For example, ?? -1, ?? 10 THRU 15, etc.

For each event specified, ?? prints the event number, the prompt, the input line(s), and the value(s). If the event input was a p.a. command that "unread" some other input lines, the p.a. command is printed without a preceding prompt, to show that they are not stored as the input, and the input lines are printed with prompts.

Events are initially stored on the history list with their value field equal to the character "bell" (control-G). Thefore, if an operation fails to complete for any reason, e.g., causes an error, is aborted, etc., ?? will print a bell as its "value".

?? commands are not entered on the history list, and so do not affect relative event numbers. In other words, an event specification of -1 typed following a ?? command will refer to the event immediately preceding the ?? command.

?? is implemented via the function PRINTHISTORY, page 8.35, which can also be called directly by the user. Printing is performed via the function SHOWPRIN2 (page 6.17), so that if the value of SYSPRETTYFLG=T, events will be prettyprinted.

UNDO *EventSpec* [Prog. Asst. Command]

Undoes the side effects of the specified events. For each event undone, UNDO prints a message: RPLACA UNDONE, REDO UNDONE etc. If nothing is undone because nothing was saved, UNDO types NOTHING SAVED. If nothing was undone because the event(s) were already undone, UNDO types ALREADY UNDONE.

If *EventSpec* is not given, UNDO searches back for the last event that contained side effects, was not undone, and itself was not an UNDO command. Note that the user can undo UNDO commands themselves by specifying the corresponding event address, e.g., UNDO -7 or UNDO UNDO.

In order to restore all pointers correctly, the user should UNDO events in the reverse order from which they were executed. For example, to undo all the side effects of the last five events, perform UNDO THRU -5, *not* UNDO FROM -5. Undoing out of order may have unforseen effects if the operations are *dependent*. For example, if the user performed (NCONC1 FOO FIE), followed by (NCONC1 FOO FUM), and then undoes the (NCONC1 FOO FIE), he will also have undone the (NCONC1 FOO FUM). If he then undoes the (NCONC1 FOO FUM), he will cause the FIE to reappear, by virtue of restoring FOO to its state before the execution of (NCONC1 FOO FUM). For more details, see page 8.23.

UNDO *EventSpec* : $x_1 \cdots x_N$ [Prog. Asst. Command]

Each x_i is a pattern that is matched to a message printed by DWIM in the event(s) specified by *EventSpec*. The side effects of the corresponding DWIM corrections, and only those side effects, are undone.

For example, if DWIM printed the message PRINTT [IN FOO] -> PRINT, then UNDO : PRINTT or UNDO : PRINT would undo the correction.

Some portions of the messages printed by DWIM are strings, e.g., the message FOO UNSAVED is printed by printing FOO and then " UNSAVED". Therefore, if the user types UNDO : UNSAVED, the DWIM correction will not be found. He should instead type UNDO : FOO or UNDO : $UNSAVED$ (<esc>UNSAVED<esc>, see R command in editor, page 17.35).

NAME *LITATOM* *EventSpec* [Prog. Asst. Command]

Saves the event(s) (including side effects) specified by *EventSpec* on the property list of *LITATOM* (under the property HISTORY). For example, NAME FOO 10 THRU 15. NAME commands are undoable.

Events saved on a litatom can be retrieved with the event specification @ *LITATOM*. For example, ?? @ FOO, REDO @ FOO, etc.

Commands defined by NAME can also be typed in directly as though they were built-in commands, e.g., FOO^{cr} is equivalent to REDO @ FOO. However, if FOO is the name of a variable, it would be evaluated, i.e., FOO^{cr} would return the value of FOO.

Commands defined by NAME can also be defined to take arguments:

NAME *LITATOM* (ARG_1 ··· ARG_N) : *EventSpec* [Prog. Asst. Command]

NAME *LITATOM* ARG_1 ··· ARG_N : *EventSpec* [Prog. Asst. Command]

The arguments ARG_i are interpreted the same as the arguments for a USE command. When *LITATOM* is invoked, the argument values are substituted for ARG_1 ··· ARG_N using the same substitution algorithm as for USE.

NAME FOO *EventSpec* is equivalent to NAME FOO : *EventSpec*. In either case, if FOO is invoked *with* arguments, an error is generated.

For example, following the event (PUTD 'FOO (COPY (GETPROP 'FIE 'EXPR))), the user types NAME MOVE FOO FIE : PUTD. Then typing MOVE TEST1 TEST2 would cause (PUTD 'TEST1 (COPY (GETPROP 'TEST2 'EXPR))) to be executed, i.e., would be equivalent to typing USE TEST1 TEST2 FOR FOO FIE IN MOVE. Typing MOVE A B C D would cause two PUTD's to be executed. Note that !'s and $'s can also be employed the same as with USE. For example, if following

```
←PREPINDEX(<MANUAL>14LISP.XGP)
←FIXFILE(<MANUAL>14LISP.XGPIDX)
```

the user performed NAME FOO 14 : -2 AND -1, then FOO 15 would perform the indicated two operations with 14 replaced by 15.

RETRIEVE *LITATOM* [Prog. Asst. Command]

Retrieves and reenters on the history list the events named by *LITATOM*. Causes an error if *LITATOM* was not named by a NAME command.

For example, if the user performs NAME FOO 10 THRU 15, and at some time later types RETRIEVE FOO, 6 *new* events will be recorded on the history list (whether or not the corresponding events have been forgotten yet). Note that RETRIEVE does *not* reexecute the events, it simply retrieves them. The user can then REDO, UNDO, FIX, etc. any or all of these events. Note that the user can combine the effects of a RETRIEVE and a subsequent history command in a single operation, e.g., REDO FOO is equivalent to RETRIEVE FOO, followed by an appropriate REDO. Actually, REDO FOO is better than RETRIEVE followed by REDO since in the latter case, the corresponding events would be entered on the history list *twice*, once for the RETRIEVE and once for the REDO. Note that UNDO FOO and ?? FOO are permitted.

BEFORE *LITATOM* [Prog. Asst. Command]

Undoes the effects of the events named by *LITATOM*.

AFTER *LITATOM* [Prog. Asst. Command]

Undoes a BEFORE *LITATOM*.

BEFORE and AFTER provide a convenient way of flipping back and forth between two states, namely the state *before* a specified event or events were executed, and that state *after* execution. For example, if the user has a complex data structure which he wants to be able to interrogate before and after certain modifications, he can execute the modifications, name the corresponding events with the NAME command, and then can turn these modifications off and on via BEFORE or AFTER commands. Both BEFORE and AFTER are no-ops if the *LITATOM* was already in the corresponding state; both generate errors if *LITATOM* was not named by a NAME command.

The alternative to BEFORE and AFTER for repeated switching back and forth involves typing UNDO, UNDO of the UNDO, UNDO of that etc. At each stage, the user would have to locate the correct event to undo, and furthermore would run the risk of that event being "forgotten" if he did not switch at least once per time-slice.

Note: Since UNDO, NAME, RETRIEVE, BEFORE, and AFTER are recorded as inputs they can be referenced by REDO, USE, etc. in the normal way. However, the user must again remember that the context in which the command is reexecuted is different than the original context. For example, if the user types NAME FOO DEFINEQ THRU COMPILE, then types ... FIE, the input that will be reread will be NAME FIE DEFINEQ THRU COMPILE as was intended, but both DEFINEQ and COMPILE, will refer to the most recent event containing those atoms, namely the event consisting of NAME FOO DEFINEQ THRU COMPILE.

ARCHIVE *EventSpec* [Prog. Asst. Command]

Records the events specified by *EventSpec* on a permanent history list. This history list can be referenced by preceding a standard event specification with @@. For example, ?? @@ prints the archived history list, REDO @@ -1 will recover the corresponding event from the archived history list and redo it, etc.

The user can also provide for automatic archiving of selected events by appropriately defining ARCHIVEFN, or by putting the property *ARCHIVE*, value T, on the event. Events that are referenced by history commands are automatically marked for archiving in this fashion (See page 8.19).

FORGET *EventSpec* [Prog. Asst. Command]

Permanently erases the record of the side effects for the events specified by *EventSpec*. If *EventSpec* is omitted, forgets side effects for entire history list.

FORGET is provided for users with space problems. For example, if the user has just performed SETs, RPLACAs, RPLACDs, PUTD, REMPROPs, etc. to release storage, the old pointers would not be garbage collected until the corresponding events age sufficiently to drop off the end of the history list and be forgotten. FORGET can be used to force immediate forgetting (of the side-effects only). FORGET is not undoable (obviously).

REMEMBER *EventSpec* [Prog. Asst. Command]

Instructs the file package to "remember" the events specified by *EventSpec*. These events will be marked as changed objects of file package type EXPRESSIONS, which can be written out via the file package command P. For example, after the user types:

```
←MOVD?(DELFILE /DELFILE)
DELFILE
←REMEMBER -1
(MOVD? (QUOTE DELFILE) (QUOTE /DELFILE))
←
```

If the user calls FILES?, MAKEFILES, or CLEANUP, the command (P (MOVD? (QUOTE DELFILE) (QUOTE /DELFILE))) will be constructed by the file package and added to the filecoms indicated by the user, unless the user has already explicitly added the corresponding expression to some P command himself.

Note that "remembering" an event like (PUTPROP 'FOO 'CLISPTYPE *EXPRESSION*) will *not* result in a (PROP CLISPTYPE FOO) command, because this will save the current (at the time of the MAKEFILE) value for the CLISPTYPE property, which may or may not be *EXPRESSION*. Thus, even if there is a PROP command which saves the CLISPTYPE property for FOO in some *FILE*COMS, remembering this event will still require a (P (PUTPROP 'FOO 'CLISPTYPE *EXPRESSION*)) command to appear.

PL *LITATOM* [Prog. Asst. Command]

"Print Property List." Prints out the property list of *LITATOM* in a nice format, with PRINTLEVEL reset to (2 . 3). For example,

```
←PL +
CLISPTYPE:  12
ACCESSFNS:  (PLUS IPLUS FPLUS)
```

PL is implemented via the function PRINTPROPS.

PB *LITATOM* [Prog. Asst. Command]

"Print Bindings." Prints the value of *LITATOM* with PRINTLEVEL reset to (2 . 3). If *LITATOM* is not bound, does not attempt spelling correction or generate an error. PB is implemented via the function PRINTBINDINGS.

PB is also a break command (page 9.5). As a break command, it ascends the stack and, for each frame in which *LITATOM* is bound, prints the frame name and value of *LITATOM*. If typed in to the programmer's assistant when not at the top level, e.g. in the editor, a lower USEREXEC, etc., PB will also ascend the stack as it does with a break. However, as a programmer's assistant command, it is primarily used to examine the top level value of a variable that may or may not be bound, or to examine a variable whose value is a large list.

; *FORM* [Prog. Asst. Command]

Allows the user to type a line of text without having the programmer's assistant process it. Useful when linked to other users, or to annotate a dribble file (page 6.12).

SHH *FORM* [Prog. Asst. Command]

Allows the user to evaluate an expression without having the programmer's assistant process it or record it on a history list. Useful when one wants to bypass a programmer's assistant command or to keep the evaluation off the history list.

EXEC [Prog. Asst. Command]
(Interlisp-10) Calls SUBSYS (page 22.21) to descend to lower exec.

Rather than start up a new fork each time the user types EXEC, the EXEC command will save the old fork handle upon return from an EXEC command, and, if the fork handle is still active, reuse it for the next EXEC command, i.e. an EXEC followed by another EXEC is equivalent to an EXEC followed by a CONTIN.

CONTIN [Prog. Asst. Command]
(Interlisp-10) Performs (SUBSYS T) to continue the last call to SUBSYS (page 22.21).

TYPE-AHEAD [Prog. Asst. Command]
A command that allows the user to type-ahead an indefinite number of inputs.

The assistant responds to TYPE-AHEAD with a prompt character of >. The user can now type in an indefinite number of lines of input, under ERRORSET protection. The input lines are saved and unread when the user exits the type-ahead loop with the command $GO (<esc>GO). While in the type-ahead loop, ?? can be used to print the type-ahead, FIX to edit the type-ahead, and $Q (<esc>Q) to erase the last input (may be used repeatedly). The TYPE-AHEAD command may be aborted by $STOP (<esc>STOP); control-E simply aborts the current line of input.

For example:

```
←TYPE-AHEAD
>SYSOUT(TEM)
>MAKEFILE(EDIT)
>BRECOMPILE((EDIT WEDIT))
>F
>$Q
\\F
>$Q
\\BRECOMPILE
>LOAD(WEDIT PROP)
>BRECOMPILE((EDIT WEDIT))
>F
>MAKEFILE(BREAK)
>LISTFILES(EDIT BREAK)
>SYSOUT(CURRENT)
>LOGOUT]
>??
     >SYSOUT(TEM)
     >MAKEFILE(EDIT)
     >LOAD(WEDIT PROP)
     >BRECOMPILE((EDIT WEDIT))
     >F
     >MAKEFILE(BREAK)
     >LISTFILES(EDIT BREAK)
     >SYSOUT(CURRENT)
     >LOGOUT]
>FIX
EDIT
*(R BRECOMPILE BCOMPL)
*P
((LOGOUT) (SYSOUT &) (LISTFILES &) (MAKEFILE &) (F) (BCOMPL &)
(LOAD &) (MAKEFILE &) (SYSOUT &))
*(DELETE LOAD)
*OK
>$GO
```

Note that type-ahead can be addressed to the compiler, since it uses LISPXREAD for input. Type-ahead can also be directed to the editor, but type-ahead to the editor and to LISPX cannot be intermixed.

The following are some useful functions and variables:

(VALUEOF *LINE*) [NLambda NoSpread Function]

An nlambda function for obtaining the value of a particular event, e.g., (VALUEOF -1), (VALUEOF ←FOO -2). The value of an event consisting of several operations is a list of the values for each of the individual operations.

Note: The value field of a history entry is initialized to bell (control-G). Thus a value of bell indicates that the corresponding operation did not complete, i.e., was aborted or caused an error (or else it returned bell).

Note: Although the input for VALUEOF is entered on the history list before VALUEOF is called, (VALUEOF -1) still refers to the value of the expression immediately before the VALUEOF input, because VALUEOF effectively backs the history list up one entry when it retrieves the specified event. Similarly, (VALUEOF FOO) will find the first event before this one that contains a FOO.

IT [Variable]

The value of the variable IT is always the value of the last event executed, i.e. (VALUEOF -1). For example,

```
←(SQRT 2)
1.414214
←(SQRT IT)
1.189207
```

If the last event was a multiple event, e.g. REDO -3 THRU -1, IT is set to value of the last of these events. Following a ?? command, IT is set to value of the last event printed. In other words, in all cases, IT is set to the last value printed on the terminal.

control-U

When typed in at any point during an input being read by LISPXREAD, permits the user to edit the input before it is returned to the calling function.

Note: control-N for Interlisp on TOPS-20.

This feature is useful for correcting mistakes noticed in typing *before* the input is executed, instead of waiting till after execution and then performing an UNDO and a FIX. For example, if the user types

"(DEFINEQ FOO (LAMBDA (X) (FIXSPELL X" and at that point notices the missing left parenthesis, instead of completing the input and allowing the error to occur, and then fixing the input, he can simply type control-U, and finish typing normally. Control-U can be typed at any point, even in the middle of an atom; it simply sets a variable checked by LISPXREAD.

When the line is finished, the editor is called on (DEFINEQ FOO (LAMBDA (X) (FIXSPELL X ···], which the user can then fix. If the user exits from the editor via OK, the (corrected) expression will be returned to whoever called LISPXREAD exactly as though it had been typed. If the user exits via STOP, the expression is returned so that it can be stored on the history list. However it will *not* be executed. In other words, the effect is the same as though the user had typed control-E at exactly the right instant.

Control-U also works for calls to READLINE (page 8.30), i.e., for line commands.

8.2.3 P.A. Commands Applied to P.A. Commands

Programmer's assistant commands that unread expressions, such as REDO, USE, etc. do not appear in the input portion of events, although they are stored elsewhere in the event. They do not interfere with or affect the searching operations of event specifications. As a result, p.a. commands themselves cannot be recovered for execution in the normal way. For example, if the user types USE A B C FOR D and follows this with USE E FOR D, he will not produce the effect of USE A B C FOR E, but instead will simply cause E to be substituted for D in the last event containing a D. To produce the desired effect, the user should type USE D FOR E IN USE. The appearance of the word REDO, USE or FIX in an event address specifies a search for the corresponding programmer's assistant command. It also specifies that the text of the programmer's assistant command itself be treated as though it were the input. However, the user must remember that the *context* in which a history command is reexecuted is that of the current history, not the original context. For example, if the user types USE FOO FOR FIE IN -1, and then later types REDO USE, the -1 will refer to the event before the REDO, not before the USE.

The one exception to the statement that programmer's assistant commands "do not interfere with or affect the searching operations of event specifications" occurs when a p.a. command fails to produce any input. For example, suppose the user types USE LOG FOR ANTILOG AND ANTILOG FOR LOGG, mispelling the second LOG. This will cause an error, LOGG ?. Since the USE command did not produce any input, the user can repair it by typing USE LOG FOR LOGG, without having to specify IN USE. This latter USE command will invoke a search for LOGG, which *will* find the bad USE command. The programmer's assistant then performs the indicated substitution, and unreads USE LOG FOR ANTILOG AND ANTILOG FOR LOG. In turn, this USE command invokes a search for ANTILOG, which, because it was not typed in but reread, ignores the bad USE command which was found by the earlier search for LOGG, and which is still on the history list. In other words, p.a. commands that fail to produce input are visible to searches arising from event specifications typed in by the user, but not to secondary event specifications.

In addition, if the most recent event is a history command which failed to produce input, a secondary event specification will effectively back up the history list one event so that relative event numbers for that event specification will not count the bad p.a. command. For example, suppose the user types USE LOG FOR ANTILOG AND ANTILOG FOR LOGG IN -2 AND -1, and after the p.a. types LOGG ?, the user types USE LOG FOR LOGG. He thus causes the command USE LOG FOR ANTILOG AND ANTILOG FOR LOG IN -2 AND -1 to be constructed and unread. In the normal case, -1 would refer to the last event, i.e., the "bad" USE command, and -2 to the event before it. However, in this case, -1 refers to the event before the bad USE command, and the -2 to the event before that. In short, the caveat above that "the user must remember that the context in which a history command is reexecuted is that of the current history, not the original context" does not apply if the correction is performed immediately.

8.3 CHANGING THE PROGRAMMER'S ASSISTANT

(CHANGESLICE *N HISTORY* —) [Function]

Changes the time-slice of the history list *HISTORY* to *N* (see page 8.25). If *HISTORY* is NIL, changes both the top level history list LISPXHISTORY and the edit history list EDITHISTORY.

Note: The effect of *increasing* the time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* the time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, CHANGESLICE is undoable, so that these events are (temporarily) recoverable. Therefore, if the user wants to recover the storage associated with these events without waiting *N* more events until the CHANGESLICE event drops off the history list, he must perform a FORGET command (page 8.13).

PROMPT#FLG [Variable]

When this variable is set to T, the current event number to be printed before each prompt character. See PROMPTCHAR, page 8.31. PROMPT#FLG is initially T.

PROMPTCHARFORMS [Variable]

The value of PROMPTCHARFORMS is a list of expression which are evaluated each time PROMPTCHAR (page 8.31) is called to print the prompt character. If PROMPTCHAR is going to print something, it first maps down PROMPTCHARFORMS evaluating each expression under an ERRORSET.

These expressions can access the special variables HISTORY (the current history list), ID (the prompt character to be printed), and PROMPTSTR, which is what PROMPTCHAR will print before ID, if anything. When PROMPT#FLG is T, PROMPTSTR will be the event number. The expressions on PROMPTCHARFORMS can change the shape of a cursor, update a clock, check for mail, etc. or change what PROMPTCHAR is about to print by resetting ID and/or PROMPTSTR. After the expressions on PROMPTCHARFORMS have been evaluated, PROMPTSTR is printed if it is (still) non-NIL, and then ID is printed, if it is (still) non-NIL.

HISTORYSAVEFORMS [Variable]

The value of HISTORYSAVEFORMS is a list of expressions that are evaluated under errorset protection each time HISTORYSAVE (page 8.32) creates a new event. This happens each time there is an interaction with the user, but not when performing an operation that is being redone.

The expressions on HISTORYSAVEFORMS are presumably executed for effect, and can access the special variables HISTORY (the current history list), ID (the current prompt character), and EVENT (the current event which HISTORYSAVE is going to return).

Note that PROMPTCHARFORMS and HISTORYSAVEFORMS together enable bracketing each interaction with the user. These can be used to measure how long the user takes to respond, to use a different readtable or terminal table, etc.

RESETFORMS [Variable]

The value of RESETFORMS is a list of forms that are evaluated at each RESET, i.e. when user types control-D, calls function RESET, or types control-C followed by START.

ARCHIVEFN [Variable]

If the *value* of ARCHIVEFN is T, and an event is about to drop off the end of the history list and be forgotten, ARCHIVEFN is called as a function with two arguments: the input portion of the event, and the entire event (see page 8.25 for the format of events). If ARCHIVEFN returns T, the event is archived on a permanent history list (see page 8.13). Note that ARCHIVEFN must be *both* set and defined. ARCHIVEFN is initially NIL and undefined.

For example, defining ARCHIVEFN as (LAMBDA (X Y) (EQ (CAR X) 'LOAD)) will keep a record of all calls to LOAD.

ARCHIVEFLG [Variable]

If the value of ARCHIVEFLG is non-NIL, the system automatically marks all events that are referenced by history commands so that they will be archived when they drop off the history list. ARCHIVEFLG is initially T, so once an event is redone, it is guaranteed to be saved.

An event is "marked for archiving" by putting the property *ARCHIVE*, value T, on the event (see page 8.25). The user could do this by means of an appropriately defined LISPXUSERFN (see below).

LISPXMACROS [Variable]

LISPXMACROS provides a macro facility that allows the user to define his own programmer's assistant commands. It is a list of elements of the form (*COMMAND DEF*). Whenever *COMMAND* appears as the first expression on a line in a LISPX input, the variable LISPXLINE is bound to the rest of the line, the event is recorded on the history list, *DEF* is evaluated, and *DEF*'s value is stored as the value of the event. Similarly, whenever *COMMAND* appears as CAR of a form in a LISPX input, the variable LISPXLINE is bound to CDR of the form, the event is recorded, and *DEF* is evaluated.

An element of the form (*COMMAND* NIL *DEF*) is interpreted to mean bind LISPXLINE and evaluate *DEF* as described above, except do *not* save the event on the history list.

LISPXHISTORYMACROS [Variable]

LISPXHISTORYMACROS allows the user to define programmer's assistant commands that re-execute other events. LISPXHISTORYMACROS is interpreted the same as LISPXMACROS, except that the result of evaluating *DEF* is treated as a list of expressions to be *unread*, exactly as though the expressions had been retrieved by a REDO command, or computed by a USE command. Note that returning NIL means nothing else is done. This provides a mechanism for defining LISPX commands which are executed for effect only.

Many programmer's assistant commands, such as RETRIEVE, BEFORE, AFTER, etc. are implemented through LISPXMACROS or LISPXHISTORYMACROS.

Note: Definitions of commands on LISPXMACROS or LISPXHISTORYMACROS can be saved on files with the file package command LISPXMACROS (see page 11.24).

LISPXUSERFN [Variable]

When LISPXUSERFN is set to T, it is applied as a function to all inputs not recognized as a programmer's assistant command, or on LISPXMACROS or LISPXHISTORYMACROS. If LISPXUSERFN decides to handle this input, it simply processes it (the event was already stored on the history list before LISPXUSERFN was called), sets LISPXVALUE to the value for the event, and returns T. The programmer's assistant will then know not to call EVAL or APPLY, and will simply store LISPXVALUE into the value slot for the event, and print it. If LISPXUSERFN returns NIL, EVAL or APPLY is called in the usual way. Note that LISPXUSERFN must be both set and defined.

LISPXUSERFN is given two arguments: *X* and *LINE*. *X* is the first expression typed, and *LINE* is the rest of the line, as read by READLINE (page 8.30). For example, if the user typed FOO(A B C), *X*=FOO, and *LINE*=((A B C)); if the user typed (FOO A B C), *X*=(FOO A B C), and *LINE*=NIL; and if the user typed FOO A B C, *X*=FOO and *LINE*=(A B C).

By appropriately defining (and setting) LISPXUSERFN, the user can with a minimum of effort incorporate the features of the programmer's assistant into his own executive (actually it is the other way around). For example, LISPXUSERFN could be defined to parse all input (other than p.a. commands) in an alternative way. Note that since LISPXUSERFN is called for each input (except for p.a. commands), it can also be used to monitor some condition or gather statistics.

(LISPXPRINT *X Y Z NODOFLG*) [Function]
(LISPXPRIN1 *X Y Z NODOFLG*) [Function]
(LISPXPRIN2 *X Y Z NODOFLG*) [Function]
(LISPXSPACES *X Y Z NODOFLG*) [Function]
(LISPXTERPRI *X Y Z NODOFLG*) [Function]
(LISPXTAB *X Y Z NODOFLG*) [Function]
(LISPXPRINTDEF *EXPR FILE LEFT DEF TAIL NODOFLG*) [Function]

In addition to saving inputs and values, the programmer's assistant saves most system messages on the history list. For example, FILE CREATED ···, (*FN* REDEFINED), (*VAR* RESET), output of TIME, BREAKDOWN, STORAGE, DWIM messages, etc. When ?? prints the event, the output is also printed. This facility is implemented via these functions.

These functions print exactly the same as their non-LISPX counterparts. Then, they put the output on the history list under the property *LISPXPRINT* (see page 8.25).

If *NODOFLG* is non-NIL, these fuctions do not print, but only put their output on the history list.

To perform output operations from user programs so that the output will appear on the history list, the program needs simply to call the corresponding LISPX printing function.

(-USERLISPXPRINT *X FILE Z NODOFLG*) [Function]

The function USERLISPXPRINT is available to permit the user to define additional LISPX printing functions. If the user has a function *FN* that takes three or fewer arguments, and the second argument is the file name, he can define a LISPX printing function by simply giving LISPX*FN* the definition of USERLISPXPRINT, for example, with MOVD(USERLISPXPRINT LISPX*FN*). USERLISPXPRINT is defined to look back on the stack, find the name of the calling function, strip off the leading "LISPX", perform the appropriate saving information, and then call the function to do the actual printing.

LISPXPRINTFLG [Variable]

If LISPXPRINTFLG=NIL, the LISPX printing functions will not store their output on the history list. LISPXPRINTFLG is initially T.

8.4 STATISTICS

The programmer's assistant keeps various statistics about system usage, e.g., number of user inputs, number of undo saves, number of calls to editor, number of edit commands, number of p.a. commands, cpu time, console time, etc. These can be viewed via the function LISPXSTATS. The user can define add new statistics to the p.a. statistics via the function ADDSTATS, and increment them with LISPXWATCH.

Note: The collection of programmer's assistant statistics is not supported in Interlisp-D. ADDSTATS and LISPXWATCH are defined with null definitions, so programs can be transferred.

(LISPXSTATS *RETURNVALUESFLG*) [Function]

Prints programmer's assistant statistics. If *RETURNVALUESFLG*=T, returns the statistics as a list of elements of the form (*VALUE* . *EXPLANATION*).

(ADDSTATS *STAT*$_1$ ··· *STAT*$_N$) [NLambda NoSpread Function]

Each *STAT*$_i$ is a list of the form (*STAT-NAME* . *MESSAGE*). Each *STAT-NAME* is defined as the name of a new statistic.

For example, (ADDSTATS (EDITCALLS CALLS TO EDITOR) (UNDOSTATS CHANGES UNDONE) will define two new statistics, named EDITCALLS and UNDOSTATS.

(LISPXWATCH *STAT N*) [Function]

Increments the statistic with name *STAT* by *N* (or 1 if *N*=NIL).

LISPXWATCH has a BLKLIBRARYDEF (see page 12.14).

The user can save his statistics for loading into a new system by performing MAKEFILE(DUMPSTATS). After the file DUMPSTATS is loaded, the statistics printed by LISPXSTATS will be the same as those that would be printed following the MAKEFILE.

8.5 UNDOING

Note: This discussion only applies to undoing under the executive and break; the editors handles undoing itself in a slightly different fashion.

The UNDO capability of the programmer's assistant is implemented by requiring that each operation that is to be undoable be responsible itself for saving on the history list enough information to enable reversal of its side effects. In other words, the assistant does not "know" when it is about to perform a destructive operation, i.e., it is not constantly checking or anticipating. Instead, it simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible functions. The UNDO command, which involves recovering the saved information and performing the corresponding inverses, works the same way, so that the user can UNDO an UNDO, and UNDO that etc.

At each point, until the user specifically requests an operation to be undone, the assistant does not know, or care, whether information has been saved to enable the undoing. Only when the user attempts to undo an operation does the assistant check to see whether any information has been saved. If none has been saved, and the user has specifically named the event he wants undone, the assistant types NOTHING SAVED. (When the user simply types UNDO, the assistant searches for the last undoable event, ignoring events already undone as well as UNDO operations themselves.)

This implementation minimizes the overhead for undoing. Only those operations which actually make changes are affected, and the overhead is small: two or three cells of storage for saving the information, and an extra function call. However, even this small price may be too expensive if the operation is sufficiently primitive and repetitive, i.e., if the extra overhead may seriously degrade the overall performance of the program. Hence not every destructive operation in a program should necessarily be undoable; the programmer must be allowed to decide each case individually.

Therefore for each primitive destructive function, Interlisp has defined an undoable version which always saves information. By convention, the name of the undoable version of a function is the function name, preceeded by "/." For example, there is RPLACA and /RPLACA, REMPROP and /REMPROP, etc. The "slash" functions that are currently implemented can be found as the value of /FNS.

The various system packages use the appropriate undoable functions. For example, BREAK uses /PUTD and /REMPROP so as to be undoable, and DWIM uses /RPLACA and /RPLACD, when it makes a correction.[2] Similarly, the user can simply use the corresponding / function if he wants to make a destructive operation in his own program undoable. When the / function is called, it will save the UNDO information in the current event on the history list.

The programmer's assistant cannot know whether efficiency and overhead are serious considerations for the execution of an expression in a user *program*, so the user must decide if he wants these operations undoable by explicitly calling /MAPCONC, etc. However, *typed-in* expressions rarely involve iterations or lengthy computations *directly*. Therefore, before evaluating the user input, the programmer's assistant substitutes the corresponding undoable function for any destructive function (see LISPX/, page 8.34). For example, if the user types (MAPCONC NASDIC ···), it is actually (/MAPCONC NASDIC ···) that is evaluated. Obviously, with a more sophisticated analysis of both user input and user programs, the

[2] The effects of the following functions are always undoable: DEFINE, DEFINEQ, DEFC (used to give a function a compiled code definition), DEFLIST, LOAD, SAVEDEF, UNSAVEDEF, BREAK, UNBREAK, REBREAK, TRACE, BREAKIN, UNBREAKIN, CHANGENAME, EDITFNS, EDITF, EDITV, EDITP, EDITE, EDITL, ESUBST, ADVISE, UNADVISE, READVISE, plus any changes caused by DWIM.

decision concerning which operations to make undoable could be better advised. However, we have found the configuration described here to be a very satisfactory one. The user pays a very small price for being able to undo what he types in, and if he wishes to protect himself from malfunctioning in his own programs, he can have his program explicitly call undoable functions.

8.5.1 Undoing Out of Order

/RPLACA operates undoably by saving (on the history list) the list cell that is to be changed and its original CAR. Undoing a /RPLACA simply restores the saved CAR. This implementation can produce unexpected results when multiple /RPLACAs are done on the same list cell, and then undone out of order. For example, if the user types (RPLACA FOO 1), followed by (RPLACA FOO 2), then undoes both events by undoing the most recent event first, then undoing the older event, FOO will be restored to its state before either RPLACA operated. However if the user undoes the first event, *then* the second event, (CAR FOO) will be 1, since this is what was in CAR of FOO before (RPLACA FOO 2) was executed. Similarly, if the user types (NCONC1 FOO 1), followed by (NCONC1 FOO 2), undoing just (NCONC1 FOO 1) will remove both 1 and 2 from FOO. The problem in both cases is that the two operations are not "independent." In general, operations are always independent if they affect different lists or different sublists of the same list. Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the "right" thing. However, undoing dependent operations out of order may not always have the predicted effect.

Property list operations, (i.e., PUTPROP, ADDPROP and REMPROP) are handled specially, so that operations that affect different properties on the same property list are always independent. For example, if the user types (PUTPROP 'FOO 'BAR 1) then (PUTPROP 'FOO 'BAZ 2), then undoes the first event, the BAZ property will remain, even though it may not have been on the property list of FOO at the time the first event was executed.

8.5.2 SAVESET

Typed-in SETs are made undoable by substituting a call to SAVESET. SETQ is made undoable by substituting SAVESETQ, and SETQQ by SAVESETQQ, both of which are implemented in terms of SAVESET.

In addition to saving enough information on the history list to enable undoing, SAVESET operates in a manner analogous to SAVEDEF (page 11.18) when it resets a top level value: when it changes a top level binding from a value other than NOBIND to a new value that is not EQUAL to the old one, SAVESET saves the old value of the variable being set on the variable's property list under the property VALUE, and prints the message (*VARIABLE* RESET). The old value can be restored via the function UNSET, which also saves the current value (but does not print a message). Thus UNSET can be used to flip back and forth between two values.

Of course, UNDO can be used as long as the event containing this call to SAVESET is still active. Note however that the old value will remain on the property list, and therefore be recoverable via UNSET, even after the original event has been forgotten.

RPAQ and RPAQQ are implemented via calls to SAVESET. Thus old values will be saved and messages printed for any variables that are reset as the result of loading a file.

For top level variables, SAVESET also adds the variable to the appropriate spelling list, thereby noticing variables set in files via RPAQ or RPAQQ, as well as those set via type-in.

(SAVESET *NAME VALUE TOPFLG FLG*) [Function]

An undoable SET. SAVESET scans the stack looking for the last binding of *NAME*, sets *NAME* to *VALUE*, and returns *VALUE*.

If the binding changed was a top level binding, *NAME* is added to the spelling list SPELLINGS3 (see page 15.14). Furthermore, if the old value was not NOBIND, and was also not EQUAL to the new value, SAVESET calls the file package to update the necessary file records. Then, if DFNFLG is not equal to T, SAVESET prints (*NAME* RESET), and saves the old value on the property list of *NAME*, under the property VALUE.

If *TOPFLG*=T, SAVESET operates as above except that it always uses *NAME*'s top-level value cell. When *TOPFLG* is T, and DFNFLG is ALLPROP and the old value was not NOBIND, SAVESET simply stores *VALUE* on the property list of *NAME* under the property VALUE, and returns *VALUE*. This option is used for loading files without disturbing the current value of variables (see page 5.9).

If *FLG*=NOPRINT, SAVESET saves the old value, but does not print the message. This option is used by UNSET.

If *FLG*=NOSAVE, SAVESET does *not* save the old value on the property list, nor does it add *NAME* to SPELLINGS3. However, the call to SAVESET is still undoable. This option is used by /SET.

If *FLG*=NOSTACKUNDO, SAVESET is undoable only if the binding being changed is a top-level binding, i.e. this says when resetting a variable that has been rebound, don't bother to make it undoable. This option is used by RPAQ, RPAQQ, and ADDTOVAR.

(UNSET *NAME*) [Function]

If *NAME* does not contain a property VALUE, UNSET generates an error. Otherwise UNSET calls SAVESET with *NAME*, the property value, *TOPFLG*=T, and *FLG*=NOPRINT.

8.5.3 UNDONLSETQ and RESETUNDO

The function UNDONLSETQ provides a limited form of backtracking: if an error occurs under the UNDONLSETQ, all undoable side effects executed under the UNDONLSETQ are undone. RESETUNDO, used in conjunction with RESETLST and RESETSAVE (page 9.19), provides a more general undo capability where the user can specify that the side effects be undone after the specified computation finishes, is aborted by an error, or by a control-D.

(UNDONLSETQ *UNDOFORM* —) [NLambda Function]

An nlambda function similar to NLSETQ (page 9.15). UNDONLSETQ evaluates *UNDOFORM*, and if no error occurs during the evaluation, returns (LIST (EVAL *UNDOFORM*)) and passes the undo information from *UNDOFORM* (if any) upwards. If an error does occur, the UNDONLSETQ returns NIL, and any undoable changes made during the evaluation of *UNDOFORM* are undone.

Any undo information is stored directly on the history event (if LISPXHIST is not NIL), so that if the user control-D's out of the UNDONLSETQ, the event is still undoable.

UNDONLSETQ will operate correctly if #UNDOSAVES is or has been exceeded for this event, or is exceeded while under the scope of the UNDONLSETQ.

Note: Caution must be exercised in using coroutines or other non-standard means of exiting while under an UNDONLSETQ. See discussion in page 9.19.

(RÉSETUNDO *X* *STOPFLG*) [Function]

For use in conjunction with RESETLST (page 9.19). (RESETUNDO) initializes the saving of undo information and returns a value which when given back to RESETUNDO undoes the intervening side effects. For example, (RESETLST (RESETSAVE (RESETUNDO)) . *FORMS*) will undo the side effects of *FORMS* on normal exit, or if an error occurs or a control-D is typed.

If *STOPFLG*=T, RESETUNDO stops accumulating undo information it is saving on *X*. Note that this has no bearing on the saving of undo information on higher RESETUNDO's, or on being able to undo the entire event.

For example,

```
(RESETLST
   (SETQ FOO (RESETUNDO))
   (RESETSAVE NIL (LIST 'RESETUNDO FOO))
   (ADVISE ···)
   (RESETUNDO FOO T)
   . FORMS)
```

would cause the advice to be undone, but *not* any of the side effects in *FORMS*.

8.6 FORMAT AND USE OF THE HISTORY LIST

The system currently uses three history lists, LISPXHISTORY for the top-level Interlisp executive, EDITHISTORY for the editors, and ARCHIVELST for archiving events (see page 8.13). All history lists have the same format, use the same functions, HISTORYSAVE, for recording events, and use the same set of functions for implementing commands that refer to the history list, e.g., HISTORYFIND, PRINTHISTORY, UNDOSAVE, etc.

Each history list is a list of the form (*L* *EVENT#* *SIZE* *MOD*), where *L* is the list of events with the most recent event first, *EVENT#* is the event number for the most recent event on *L*, *SIZE* is the size of the time-slice (below), i.e., the maximum length of *L*, and *MOD* is the highest possible event number. LISPXHISTORY and EDITHISTORY are both initialized to (NIL 0 100 100). Setting LISPXHISTORY or EDITHISTORY to NIL disables all history features, so LISPXHISTORY and EDITHISTORY act like flags as well as repositories of events.

Each history list has a maximum length, called its "time-slice." As new events occur, existing events are aged, and the oldest events are "forgotten." For efficiency, the storage used to represent the forgotten event is reused in the representation of the new event, so the history list is actually a ring buffer. The time-slice of a history list can be changed with the function CHANGESLICE, page 8.18. Larger time-slices enable longer "memory spans," but tie up correspondingly greater amounts of storage. Since the user seldom needs really "ancient history," and a facility is provided for saving and remembering selected events (see NAME and RETRIEVE, page 8.12), a relatively small time-slice such as 30 events is more than adequate, although some users prefer to set the time-slice as large as 100 events.

If PROMPT#FLG (page 8.18) is set to T, an "event number" will be printed before each prompt. More recent events have higher numbers. When the event number of the current event is 100, the next event will be given number 1. If the time-slice is greater than 100, the "roll-over" occurs at the next highest hundred, so that at no time will two events ever have the same event number. For example, if the time-slice is 150, event number 1 will follow event number 200.

Each individual event on *L* is a list of the form (*INPUT ID VALUE . PROPS*). *ID* is the prompt character for this event, e.g., ←, :, *, etc. *VALUE* is the value of the event, and is initialized to bell.[3] *PROPS* is a property list used to associate other information with the event (described below).

INPUT is the input sequence for the event. Normally, this is just the input that the user typed-in. For an APPLY format input, this is a list consisting of two expressions; for an EVAL format input, this is a list of just one expression; for an input entered as list of atoms, *INPUT* is simply that list. For example,

User Input	*INPUT* is:
PLUS[1 1]	(PLUS (1 1))
(PLUS 1 1)	((PLUS 1 1))
PLUS 1 1^{cr}	(PLUS 1 1)

If the user types in a programmer's assistant command that "unreads" and reexecutes other events (REDO, USE., etc.), *INPUT* contains a "sequence" of the inputs from the redone events. Specifically, the *INPUT* fields from the specified events are concatenated into a single list, seperated by special markers called "pseudo-carriage returns," which print out as the string "<c.r.>".[4] When the result of this concatenation is "reread," the pseudo-carriage-returns are treated by LISPXREAD and READLINE exactly as real carriage returns, i.e., they serve to distinguish between APPLY and EVAL formats on inputs to LISPX, and to delimit line commands to the editor.

The same convention is used for representing multiple inputs when a USE command involves sequential substitutions. For example, if the user types GETD(FOO) and then USE FIE FUM FOR FOO, the input sequence that will be constructed is (GETD (FIE) "<c.r.>" GETD (FUM)), which is the result of substituting FIE for FOO in (GETD (FOO)) concatenated with the result of substituting FUM for FOO in (GETD (FOO)).

Note that once a multiple input has been entered as the input portion of a new event, that event can be treated exactly the same as one resulting from type-in. In other words, no special checks have to be made when *referencing* an event, to see if it is simple or multiple. This implementation permits an

[3] On EDITHISTORY, this field is used to save the side effects of each command. See page 8.35.

[4] The value of the variable HISTSTR0 is used to represent a pseudo-carriage return. This is initially the string "<c.r.>". Note that the functions that recognize pseudo-carriage returns compare them to HISTSTR0 using EQ, so this marker will never be confused with a string that was typed in by the user.

event specification to refer to a single simple event, or to several events, or to a single event originally constructed from several events (which may themselves have been multiple input events, etc.) without having to treat each case separately.

REDO, RETRY, USE, ..., and FIX commands, i.e., those commands that reexecute previous events, are not stored as inputs, because the input portion for these events are the expressions to be "reread". The history commands UNDO, NAME, RETRIEVE, BEFORE, and AFTER *are* recorded as inputs, and ?? prints them exactly as they were typed.

PROPS is a property list of the form ($PROPERTY_1$ $VALUE_1$ $PROPERTY_2$ $VALUE_2$ ···), that can be used to associate arbitrary information with a particular event. Currently, the following properties are used by the programmer's assistant:

SIDE — A list of the side effects of the event. See UNDOSAVE, page 8.33.

PRINT — Used by the ?? command when special formatting is required, for example, when printing events corresponding to the break commands OK, GO, EVAL, and ?=.

USE-ARGS
...ARGS — The USE-ARGS and ...ARGS properties are used to save the arguments and expression for the corresponding history command.

ERROR
CONTEXT — *ERROR* and *CONTEXT* are used to save information when errors occur for subsequent use by the $ command. Whenever an error occurs, the offender is automatically saved on that event's entry in the history list, under the *ERROR* property.

LISPXPRINT — Used to record calls to LISPXPRINT, LISPXPRIN1, etc. (see page 8.20).

ARCHIVE — The property *ARCHIVE* on an event causes the event to be automatically archived when it "falls off the end" of the history list (see page 8.13).

GROUP
HISTORY — The *HISTORY* and *GROUP* properties are used for commands that reexecute previous events, i.e., REDO, RETRY, USE, ..., and FIX. The value of the *HISTORY* property is the history command that the user actually typed, e.g., REDO FROM F. This is used by the ?? command when printing the event. The value of the *GROUP* property is a structure containing the side effects, etc. for the individual inputs being reexecuted. This structure is described below.

When LISPX is given an input, it calls HISTORYSAVE (page 8.32) to record the input in a new event.[5] Normally, HISTORYSAVE creates and returns a new event. LISPX binds the variable LISPXHIST to the value of HISTORYSAVE, so that when the operation has completed, LISPX knows where to store the value. Note that by the time it completes, the operation may no longer correspond to the most recent event on the history list. For example, all inputs typed to a lower break will appear later on the

[5] The commands ??, FORGET, TYPE-AHEAD, $BUFS, and ARCHIVE are executed immediately, and are not recorded on the history list.

history list. After binding LISPXHIST, LISPX executes the input, stores its value in the value field of the LISPXHIST event, prints the value, and returns.

When the input is a REDO, RETRY, USE, . . ., or FIX command, the procedure is similar, except that the event is also given a *GROUP* property, initially NIL, and a *HISTORY* property, and LISPX simply unreads the input and returns. When the input is "reread", it is HISTORYSAVE, not LISPX, that notices this fact, and finds the event from which the input originally came.[6] HISTORYSAVE then adds a new (*INPUT ID VALUE . PROPS*) entry to the *GROUP* property for this event, and returns this entry as the "new event." LISPX then proceeds exactly as when its input was typed directly, i.e., it binds LISPXHIST to the value of HISTORYSAVE, executes the input, stores the value in CADDR of LISPXHIST, prints the value, and returns. In fact, LISPX never notices whether it is working on freshly typed input, or input that was reread. Similarly, UNDOSAVE will store undo information on LISPXHIST the same as always, and does not know or care that LISPXHIST is not the entire event, but one of the elements of the *GROUP* property. Thus when the event is finished, its entry will look like:

```
(INPUT ID VALUE
    *HISTORY*
         COMMAND
    *GROUP*
         ((INPUT1 ID1 VALUE1 SIDE SIDE1)
          (INPUT2 ID2 VALUE2 SIDE SIDE2)
           ...))
```

In this case, the value field of the event with the *GROUP* property is not being used; VALUEOF instead returns a list of the values from the *GROUP* property. Similarly, UNDO operates by collecting the SIDE properties from each of the elements of the *GROUP* property, and then undoing them in reverse order.

This implementation removes the burden from the function calling HISTORYSAVE of distinguishing between new input and reexecution of input whose history entry has already been set up.

8.7. PROGRAMMER'S ASSISTANT FUNCTIONS

(LISPX *LISPXX LISPXID LISPXXMACROS LISPXXUSERFN LISPXFLG*) [Function]

LISPX is the primary function of the programmer's assistant. LISPX takes one user input, saves it on the history list, evaluates it, saves its value, and prints and returns it. LISPX also interpretes p.a. commands, LISPXMACROS, LISPXHISTORYMACROS, and LISPXUSERFN.

If *LISPXX* is a list, it is interpreted as the input expression. Otherwise, LISPX calls READLINE, and uses *LISPXX* plus the value of READLINE as the input for the event. If *LISPXX* is a list CAR of which is LAMBDA or NLAMBDA, LISPX calls LISPXREAD to obtain the arguments.

LISPXID is the prompt character to print before accepting user input. A user can call LISPX specifying any prompt character as *LISPXID* except for *, since in

[6] If HISTORYSAVE cannot find the event, for example if a user program unreads the input directly, and not via a history command, HISTORYSAVE proceeds as though the input were typed.

certain cases LISPX must use the value of *LISPXID* to tell whether or not it was called from the editor.

If *LISPXXMACROS* is not NIL, it is used as the list of LISPX macros, otherwise the top level value of the variable LISPXMACROS is used.

If *LISPXXUSERFN* is not NIL, it is used as the LISPXUSERFN. In this case, it is not necessary to both set and define LISPXUSERFN as described on page 8.20.

LISPXFLG is used by the E command in the editor (see page 8.35).

Note that the history is *not* one of the arguments to LISPX, i.e., the editor must bind (reset) LISPXHISTORY to EDITHISTORY before calling LISPX to carry out a history command. LISPX will continue to operate as an EVAL/APPLY function if LISPXHISTORY is NIL. Only those functions and commands that involve the history list will be affected.

LISPX performs spelling corrections using LISPXCOMS, a list of its commands, as a spelling list whenever it is given an unbound atom or undefined function, before attempting to evaluate the input.

LISPX is responsible for rebinding HELPCLOCK, used by BREAKCHECK (page 9.10) for computing the amount of time spent in a computation, in order to determine whether to go into a break if and when an error occurs.

(USEREXEC *LISPXID LISPXXMACROS LISPXXUSERFN*) [Function]

Repeatedly calls LISPX under errorset protection specifying *LISPXXMACROS* and *LISPXXUSERFN*, and using *LISPXID* (or ← if *LISPXID*=NIL) as a prompt character. USEREXEC is exited via the command OK, or else with a RETFROM.

(LISPXEVAL *LISPXFORM LISPXID*) [Function]

Evaluates *LISPXFORM* (using EVAL) the same as though it were typed in to LISPX, i.e., the event is recorded, and the evaluation is made undoable by substituting the slash functions for the corresponding destructive functions (see page 8.22). LISPXEVAL returns the value of the form, but does not print it.

When LISPX recieves an "input," it may come from the user typing it in, or it may be an input that has been "unread." LISPX handles these two cases by getting inputs with LISPXREAD and READLINE, described below. These functions use the variable READBUF to store the expressions that have been unread. When READBUF is not NIL, READLINE and LISPXREAD "read" expressions from READBUF until READBUF is NIL, or until they read a pseudo-carriage return (see page 8.26). Both functions return a list of the expressions that have been "read." (The pseudo-carriage return is not included in the list.)

When READBUF is NIL, both LISPXREAD and READLINE actually obtain their input by performing (APPLY* LISPXREADFN *FILE*), where LISPXREADFN is initially set to READ. The user can make LISPX, the editor, break, etc. do their reading via a different input function by simply setting LISPXREADFN to the name of that function (or an appropriate LAMBDA expression).

Note: The user should only add expressions to READBUF using the function LISPXUNREAD (page 8.31), which knows about the format of READBUF.

(READLINE *RDTBL* — —) [Function]

Reads a line from the terminal, returning it as a list. If (READP T) is NIL, READLINE returns NIL. Otherwise it reads expressions by performing (APPLY* LISPXREADFN T) (LISPXREADFN is initially set to READ) until it encounters either:

- a carriage-return (typed by the user) that is not preceded by any spaces, e.g.,

```
A B C^cr
```

and READLINE returns (A B C)

- a list terminating in a "]", in which case the list is included in the value of READLINE, e.g.,

```
A B (C D]
```

and READLINE returns (A B (C D)).

- an unmatched right parentheses or right square bracket, which is not included in the value of READLINE, e.g.,

```
A B C]
```

and READLINE returns (A B C).

In the case that one or more spaces precede a carriage-return, or a list is terminated with a ")", READLINE will type "..." and continue reading on the next line, e.g.,

```
A B C^cr
...(D E F)
...(X Y Z]
```

and READLINE returns (A B C (D E F) (X Y Z)).

If the user types another carriage-return after the "...", the line will terminate, e.g.,

```
A B C^cr
...^cr
```

and READLINE returns (A B C).

Note that carriage-return, i.e., the EOL character, can be redefined with SETSYNTAX (page 6.34). READLINE actually checks for the EOL character, whatever that may be. The same is true for right parenthesis and right bracket.

When READLINE is called from LISPX, it operates differently in two respects:

(1) If the line consists of a single) or], READLINE returns (NIL) instead of NIL, i.e., the) or] *is* included in the line. This permits the user to type FOO) or FOO], meaning call the function FOO with no arguments, as opposed to FOO^{cr}

(FOO<carriage-return>), meaning evaluate the variable FOO.

(2) If the first expression on the line is a list that is not preceded by any spaces, the list terminates the line regardless of whether or not it is terminated by]. This permits the user to type EDITF(FOO) as a single input.

Note that if any spaces are inserted between the atom and the left parentheses or bracket, READLINE will assume that the list does not terminate the line. This is to enable the user to type a line command such as USE (FOO) FOR FOO. Therefore, if the user accidentially puts an extra space between a function and its arguments, he will have to complete the input with another carriage return, e.g.,

```
←EDITF (FOO)
...cr
EDIT
*
```

(LISPXREAD *FILE RDTBL*) [Function]

A generalized READ. If READBUF=NIL, LISPXREAD performs (APPLY* LISPXREADFN *FILE*), which it returns as its value. If READBUF is not NIL, LISPXREAD "reads" and returns the next expression on READBUF.

Note: If the user types control-U during the call to READ, LISPXREAD calls the editor and returns the edited value.

LISPXREAD also sets REREADFLG to NIL when it reads via READ, and sets REREADFLG to the value of READBUF when rereading.

(LISPXREADP *FLG*) [Function]

A generalized READP. If *FLG*=T, LISPXREADP returns T if there is any input waiting to be "read", in the manner of LISPXREAD. If *FLG*=NIL, LISPXREADP returns T only if there is any input waiting to be "read" *on this line.* In both cases, leading spaces are ignored, i.e., skipped over with READC, so that if only spaces have been typed, LISPXREADP will return NIL.

(LISPXUNREAD *LST* —) [Function]

Unreads *LST*, a list of expressions.

(PROMPTCHAR *ID FLG HISTORY*) [Function]

Called by LISPX to print the prompt character *ID* before each input. PROMPTCHAR will not print anything when the next input will be "reread", i.e., when READBUF is not NIL.

PROMPTCHAR will not print when (READP)=T, unless *FLG* is T. The editor calls PROMPTCHAR with *FLG*=NIL so that extra *'s are not printed when the user types several commands on one line. However, EVALQT calls PROMPTCHAR with *FLG*=T, since it always wants the ← printed (except when "rereading").

If PROMPT#FLG (page 8.18) is T and *HISTORY* is not NIL, PROMPTCHAR prints the current event number (of *HISTORY*) before printing *ID*.

The value of PROMPTCHARFORMS (page 8.18) is a list of expressions that are evaluated by PROMPTCHAR before, and if, it does any printing.

(HISTORYSAVE *HISTORY ID INPUT1 INPUT2 INPUT3 PROPS*) [Function]

Records one event on *HISTORY*.

If *INPUT1* is not NIL, the input is of the form (*INPUT*$_1$ *INPUT*$_2$. *INPUT*$_3$). If *INPUT*$_1$ is NIL, and *INPUT*$_2$ is not NIL, the input is of the form (*INPUT*$_2$. *INPUT*$_3$). Otherwise, the input is just *INPUT*$_3$.

HISTORYSAVE creates a new event with the corresponding input, *ID*, value field initialized to bell, and *PROPS*. If the *HISTORY* has reached its full size, the last event is removed and cannibalized.

The value of HISTORYSAVE is the new event. However, if REREADFLG is not NIL, and the most recent event on the history list contains the history command that produced this input, HISTORYSAVE does not create a new event, but simply adds an (*INPUT ID* bell . *PROPS*) entry to the *GROUP* property for that event and returns that entry. See discussion on page 8.28.

HISTORYSAVEFORMS (page 8.18) is a list of expressions that are evaluated under errorset protection each time HISTORYSAVE creates a new event.

(LISPXSTOREVALUE *EVENT VALUE*) [Function]

Used by LISPX for storing the value of an event. Can be advised by user to watch for particular values or perform other monitoring functions.

(LISPXFIND *HISTORY LINE TYPE BACKUP* —) [Function]

LINE is an event specification, *TYPE* specifies the format of the value to be returned by LISPXFIND, and can be either ENTRY, ENTRIES, COPY, COPIES, INPUT, or REDO. LISPXFIND parses *LINE*, and uses HISTORYFIND to find the corresponding events. LISPXFIND then assembles and returns the appropriate structure.

LISPXFIND incorporates the following special features:

(1) if *BACKUP*=T, LISPXFIND interprets *LINE* in the context of the history list *before* the current event was added. This feature is used, for example, by VALUEOF, so that (VALUEOF -1) will not refer to the VALUEOF event itself.

(2) if *LINE*=NIL and the last event is an UNDO, the next to the last event is taken. This permits the user to type UNDO followed by REDO or USE.

(3) LISPXFIND recognizes @@, and substitutes ARCHIVELST for *HISTORY* (see page 8.13).

(4) LISPXFIND recognizes @, and retrieves the corresponding event(s) from the property list of the atom following @ (see page 8.12).

(HISTORYFIND *LST INDEX MOD EVENTADDRESS* —) [Function]

Searches *LST* and returns the tails of *LST* beginning with the event corresponding to *EVENTADDRESS*. *LST*, *INDEX*, and *MOD* are the first three elements of a "history list" structure (see page 8.25). *EVENTADDRESS* is an event address (see page 8.5) e.g., (43), (-1), (FOO FIE), (LOAD ← FOO), etc. If HISTORYFIND cannot find *EVENTADDRESS*, it generates an error.

(HISTORYMATCH *INPUT PAT EVENT*) [Function]

Used by HISTORYFIND for "matching" when *EVENTADDRESS* specifies a pattern. Matches *PAT* against *INPUT*, the input portion of the history event *EVENT*, as matching is defined on page 17.13. Initially defined as (EDITFINDP *INPUT PAT* T), but can be advised or redefined by the user.

(ENTRY# *HIST X*) [Function]

HIST is a history list (see page 8.25). *X* is EQ to one of the events on *HIST*. ENTRY# returns the event number for *X*.

(UNDOSAVE *UNDOFORM HISTENTRY*) [Function]

UNDOSAVE adds the "undo information" *UNDOFORM* to the SIDE property of the history event *HISTENTRY*. If there is no SIDE property, one is created. If the value of the SIDE property is NOSAVE, the information is not saved.

HISTENTRY specifies an event. If *HISTENTRY*=NIL, the value of LISPXHIST is used. If both *HISTENTRY* and LISPXHIST are NIL, UNDOSAVE is a no-op. Note that *HISTENTRY* (or LISPXHIST) can either be a "real" event, or an event within the *GROUP* property of another event (see page 8.28).

The form of *UNDOFORM* is (*FN* . *ARGS*).[7] Undoing is done by performing (APPLY (CAR *UNDOFORM*) (CDR *UNDOFORM*)). For example, if the definition of FOO is *DEF*, (/PUTD FOO *NEWDEF*) will cause a call to UNDOSAVE with *UNDOFORM*=(/PUTD FOO *DEF*).

CAR of the SIDE property of an event is a count of the number of *UNDOFORM*s saved for this event. Each call to UNDOSAVE increments this count. If this count is set to -1, then it is never incremented, and any number of *UNDOFORM*s can be saved. If this count is a positive number, UNDOSAVE restricts the number of *UNDOFORM*s saved to the value of #UNDOSAVES, described below. LOAD initializes the count to -1, so that regardless of the value of #UNDOSAVES, no message will be printed, and the LOAD will be undoable.

#UNDOSAVES [Variable]

The value of #UNDOSAVES is the maximum number of *UNDOFORM*s to be saved for a single event. When the count of *UNDOFORM*s reaches this number, UNDOSAVE prints the message CONTINUE SAVING?, asking the user if he wants to continue saving. If the user answers NO or defaults, UNDOSAVE discards the previously saved information for this event, and makes NOSAVE be the value of the property SIDE, which disables any further saving for this event. If the user answers YES, UNDOSAVE changes the count to -1, which is then never incremented, and continues saving. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone.

If #UNDOSAVES is negative, then when the count reaches -#UNDOSAVES, UNDOSAVE simply stops saving without printing any messages or interacting with the

[7] In the special case of /RPLNODE and /RPLNODE2, the format of *UNDOFORM* is (*X OLDCAR* . *OLDCDR*). When *UNDOFORM* is undone, this form is recognized and handled specially. This implementation saves space.

user. #UNDOSAVES=NIL is equivalent to #UNDOSAVES=infinity. #UNDOSAVES is initially NIL.

(NEW/FN *FN*) [Function]

NEW/FN performs the necessary housekeeping operations to make *FN* be translated to the undoable version /*FN* when typed-in. For example, RADIX can be made undoable when typed-in by performing:

```
← (DEFINEQ (/RADIX (X)
          (UNDOSAVE (LIST '/RADIX (RADIX X))
(/RADIX)
← (NEW/FN 'RADIX)
```

(LISPX/ *X FN VARS*) [Function]

LISPX/ performs the substitution of / functions for destructive functions that are typed-in. If *FN* is not NIL, it is the name of a function, and *X* is its argument list. If *FN* is NIL, *X* is a form. In both cases, LISPX/ returns *X* with the appropriate substitutions. *VARS* is a list of bound variables (optional).

LISPX/ incorporates information about the syntax and semantics of Interlisp expressions. For example, it does not bother to make undoable operations involving variables bound in *X*. It does not perform substitution inside of expressions CAR of which is an nlambda function (unless CAR of the form has the property INFO value EVAL, see page 5.4). For example, (BREAK PUTD) typed to LISPX, will break on PUTD, not /PUTD. Similarly, substitution *should* be performed in the arguments for functions like MAPC, RPTQ, etc., since these contain expressions that will be evaluated or applied. For example, if the user types (MAPC '(FOO1 FOO2 FOO3) 'PUTD) the PUTD must be replaced by /PUTD.

(UNDOLISPX *LINE*) [Function]

LINE is an event specification. UNDOLISPX is the function that executes UNDO commands by calling UNDOLISPX1 on the appropriate entry(s).

(UNDOLISPX1 *EVENT FLG* —) [Function]

Undoes one event. UNDOLISPX1 returns NIL if there is nothing to be undone. If the event is already undone, UNDOLISPX1 prints ALREADY UNDONE and returns T. Otherwise, UNDOLISPX1 undoes the event, prints a message, e.g., SETQ UNDONE, and returns T.

If *FLG*=T and the event is already undone, or is an undo command, UNDOLISPX1 takes no action and returns NIL. UNDOLISPX uses this option to search for the last event to undo. Thus when *LINE*=NIL, UNDOLISPX simply searches history until it finds an event for which UNDOLISPX1 returns T.

Undoing an event consists of mapping down (CDR of) the property value for SIDE, and for each element, applying CAR to CDR, and then marking the event undone by attaching (with /ATTACH) a NIL to the front of its SIDE property. Note that the undoing of each element on the SIDE property will usually cause undosaves to be added to the *current* LISPXHIST, thereby enabling the effects of UNDOLISPX1 to be undone.

(PRINTHISTORY *HISTORY LINE SKIPFN NOVALUES FILE*) [Function]

LINE is an event specification. PRINTHISTORY prints the events on *HISTORY* specified by *LINE*, e.g., (-1 THRU -10). Printing is performed via the function SHOWPRIN2, so that if the value of SYSPRETTYFLG=T, events will be prettyprinted.

SKIPFN is an (optional) functional argument that is applied to each event before printing. If it returns non-NIL, the event is skipped, i.e., not printed.

If *NOVALUES*=T, or *NOVALUES* applied to the corresponding event is true, the value is not printed. For example, *NOVALUES* is T when printing events on EDITHISTORY.

For example, the following LISPXMACRO will define ??' as a command for printing the history list while skipping all "large events" and not printing any values.

```
(??' (PRINTHISTORY
        LISPXHISTORY
        LISPXLINE
        (FUNCTION (LAMBDA (X)
           (IGREATERP (COUNT (CAR X)) 5)))
        T
        T))
```

8.8 THE EDITOR AND THE PROGRAMMER'S ASSISTANT

As mentioned earlier, all of the remarks concerning "the programmer's assistant" apply equally well to user interactions with EVALQT, BREAK or the editor. The differences between the editor's implementation of these features and that of LISPX are mostly obvious or inconsequential. However, for completeness, this section discusses the editor's implementation of the programmer's assistant.

The editor uses PROMPTCHAR to print its prompt character, and LISPXREAD, LISPXREADP, and READLINE for obtaining inputs. When the editor is given an input, it calls HISTORYSAVE to record the input in a new event on its history list, EDITHISTORY.[8] EDITHISTORY follows the same conventions and format as LISPXHISTORY. However, since edit commands have no value, the editor uses the value field for saving side effects, rather than storing them under the property SIDE.

The editor recognizes and processes the four commands DO, !E, !F, and !N which refer to previous events on EDITHISTORY. The editor also processes UNDO itself, as described below. All other history

[8] Except that the atomic commands OK, STOP, SAVE, P, ?, PP and E are not recorded. In addition, number commands are grouped together in a single event. For example, 3 3 -1 is considered as one command for changing position.

commands[9] are simply given to LISPX for execution, after first binding (resetting) LISPXHISTORY to EDITHISTORY. The editor also calls LISPX when given an E command (page 17.45). In this case, the editor uses the fifth argument to LISPX, *LISPXFLG*, to specify that any history commands are to be executed by a recursive call to LISPX, rather than by unreading. For example, if the user types E REDO in the editor, he wants the last event on LISPXHISTORY processed as LISPX input, and not to be unread and processed by the editor.

The major implementation difference between the editor and LISPX occurs in undoing. EDITHISTORY is a list of only the last *N* commands, where *N* is the value of the time-slice. However the editor provides for undoing *all* changes made in a single editing session, even if that session consisted of more than *N* edit commands. Therefore, the editor saves undo information independently of the EDITHISTORY on a list called UNDOLST, (although it also stores each entry on UNDOLST in the field of the corresponding event on EDITHISTORY.) Thus, the commands UNDO, !UNDO, and UNBLOCK, are not dependent on EDITHISTORY, and in fact will work if EDITHISTORY=NIL, or even in a system which does not contain LISPX at all. For example, UNDO specifies undoing the last command on UNDOLST, even if that event no longer appears on EDITHISTORY. The only interaction between UNDO and the history list occurs when the user types UNDO followed by an event specification. In this case, the editor calls LISPXFIND to find the event, and then undoes the corresponding entry on UNDOLST. Thus the user can only undo a *specified* command within the scope of the EDITHISTORY. (Note that this is also the only way UNDO commands themselves can be undone, that is, by using the history feature, to specify the corresponding event, e.g., UNDO UNDO.)

The implementation of the actual undoing is similar to the way it is done in LISPX: each command that makes a change in the structure being edited does so via a function that records the change on a variable. After the command has completed, this variable contains a list of all the pointers that have been changed and their original contents. Undoing that command simply involves mapping down that list and restoring the pointers.

[9] as indicated by their appearance on HISTORYCOMS, a list of the history commands. EDITDEFAULT interrogates HISTORYCOMS before attempting spelling correction. (All of the commands on HISTORYCOMS are also on EDITCOMSA and EDITCOMSL so that they can be corrected if misspelled in the editor.) Thus if the user defines a LISPXMACRO and wishes it to operate in the editor as well, he need simply add it to HISTORYCOMS. For example, RETRIEVE is implemented as a LISPXMACRO and works equally well in LISPX and the editor.

CHAPTER 9

ERRORS AND BREAK HANDLING

Occasionally, while a program is running, an error may occur which will stop the computation. A coding mistake may have caused the wrong arguments to be passed to a function, or the programmer may have not forseen a particular unusual situation which came up, causing a function to try doing something illegal. Interlisp provides extensive facilities for detecting and handling error conditions, to enable testing, debugging,and revising of imperfect programs.

Errors can be caused in different ways. As mentioned above, an Interlisp primitive-function may signal an error if given illegal arguments; for example, PLUS will cause an error if its arguments are not numbers. It is also possible to interrupt a computation at any time by typing one of the "interrupt characters," such as control-D or control-E (the Interlisp-D interrupt characters are listed on page 18.1; those for Interlisp-10 on page 22.1). Finally, as an aid to debugging, the programmer can specify that certain functions should cause an error automatically whenever they are entered (see page 10.1). This allows examination of the context within the computation.

When an error occurs, the system can either[1] reset and unwind the stack, or go into a "break", an environment where the user can examine the state of the system at the point of the error, and attempt to debug the program. Within a break, Interlisp offers an extensive set of "break commands", which assist with debugging.

This chapter explains what happens when errors occur. Breaks and break commands are given which allow the user to handle program errors. Finally, advanced facilities for modifying and extending the error mechanism are presented.

9.1 BREAKS

One of the most useful debugging facilities in Interlisp is the ability to put the system into a "break", stopping a computation at any point and allowing the user to interrogate the state of the world and affect the course of the computation. A break appears to the user like a top-level executive, except that a break uses the prompt character ":" to indicate it is ready to accept input(s), in the same way that "←" is used at the top-level. However, a break saves the environment where the break occurred, so that the user may evaluate variables and expressions in the environment that was broken. In addition, the break program recognizes a number of useful "break commands", which provide an easy way to interrogate the state of the broken computation.

Note: In Interlisp-D, the break package has been extended to include window operations (see page 20.10).

[1] The mechanism used for deciding whether to unwind the stack or to go into a break is described on page 9.10. The user can modify this mechanism.

Breaks may be entered in several different ways. Some interrupt characters (page 9.17) automatically cause a break to be entered whenever they are typed. Functions errors may also cause a break, depending on the depth of the computation (see page 9.10). Finally, Interlisp provides functions which make it easy to "break" suspect functions so that they always cause a break whenever they are entered, to allow examination and debugging (see page 10.4).

Within a break the user has access to all of the power of Interlisp; he can do anything that he can do at the top-level executive. For example, the user can evaluate an expression, see that the value is incorrect, call the editor, change the function, and evaluate the expression again, all without leaving the break. The user can even type in commands to the programmer's assistant (page 8.1), e.g. to redo or undo previously executed events, including break commands.

Similarly, the user can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In short, anything that he can do at the top level can be done while inside of the break. In addition the user can examine the stack (see page 7.1), and even force a return back to some higher function via the function RETFROM or RETEVAL.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction from him. If the user types in an expression whose evaluation causes an error, the break is maintained. Similarly if the user aborts a computation initiated from within the break (by typing control-E), the break is maintained. Only if the user gives one of the commands that exits from the break, or evaluates a form which does a RETFROM or RETEVAL back out of BREAK1, will the computation continue.[2]

The basic function of the break package is BREAK1. Note that BREAK1 is just another Interlisp function, not a special system feature like the interpreter or the garbage collector.It has arguments, and returns a value, the same as any other function. The value returned by BREAK1 is called "the value of the break." The user can specify this value explicitly by using the RETURN command described below. But in most cases, the value of a break is given implicitly, via a GO or OK command, and is the result of evaluating "the break expression," BRKEXP, which is one of the arguments to BREAK1. For more information on the function BREAK1, see page 9.11.

The break expression, stored in the variable BRKEXP, is an expression equivalent to the computation that would have taken place had no break occurred. For example, if the user breaks on the function FOO, the break expression is the body of the definition of FOO. When the user types OK or GO, the body of FOO is evaluated, and its value returned as the value of the break, i.e., to whatever function called FOO. BRKEXP is set up by the function that created the call to BREAK1. For functions broken with BREAK or TRACE, BRKEXP is equivalent to the body of the definition of the broken function (see page 10.4). For functions broken with BREAKIN, using BEFORE or AFTER, BRKEXP is NIL. For BREAKIN AROUND, BRKEXP is the indicated expression (see page 10.5).

BREAK1 recognizes a large set of break commands. These are typed in *without* parentheses. In order to facilitate debugging of programs that perform input operations, the carriage return that is typed to

[2] Except that BREAK1 does not "turn off" control-D, i.e., a control-D will force an immediate return back to the top level.

complete the GO, OK, EVAL, etc. commands is discarded by BREAK1, so that it will not be part of the input stream after the break.

GO [Break Command]

Evaluates BRKEXP, prints this value, and returns it as the value of the break. Releases the break and allows the computation to proceed.

OK [Break Command]

Same as GO except that the value of BRKEXP is not printed.

EVAL [Break Command]

Same as OK except that the break is maintained after the evaluation. The value of this evaluation is bound to the local variable !VALUE, which the user can interrogate. Typing GO or OK following EVAL will not cause BRKEXP to be reevaluated, but simply return the value of !VALUE as the value of the break. Typing another EVAL will cause reevaluation. EVAL is useful when the user is not sure whether the break will produce the correct value and wishes to examine it before continuing with the computation.

RETURN *FORM* [Break Command]

FORM is evaluated, and returned as the value of the break. For example, one could use the EVAL command and follow this with RETURN (REVERSE !VALUE).

↑ [Break Command]

Calls ERROR! and aborts the break, making it "go away" without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the GO, OK, EVAL, and RETURN commands, maintain the break.

The following four commands refer to "the broken function." This is the function that caused the break, whose name is stored in the BREAK1 argument BRKFN.

!EVAL [Break Command]

The broken function is first unbroken, then the break expression is evaluated (and the value stored in !VALUE), and then the function is rebroken. This command is very useful for dealing with recursive functions.

!GO [Break Command]

Equivalent to !EVAL followed by GO. The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited with the value typed.

!OK [Break Command]

Equivalent to !EVAL followed by OK. The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited.

UB [Break Command]

Unbreaks the broken function.

@ [Break Command]

Resets the variable LASTPOS, which establishes a context for the commands ?=, ARGS, BT, BTV, BTV*, EDIT, and IN? described below. LASTPOS is the position of a function call on the stack. It is initialized to the function just before the call to BREAK1, i.e., (STKNTH -1 'BREAK1).[3]

@ treats the rest of the teletype line as its argument(s). It first resets LASTPOS to (STKNTH -1 'BREAK1) and then for each atom on the line, @ searches down the stack for a call to that atom. The following atoms are treated specially:

@ Do not reset LASTPOS to (STKNTH -1 'BREAK1) but leave it as it was, and continue searching from that point.

a number *N*
If negative, move LASTPOS down the stack *N* frames. If positive, move LASTPOS up the stack *N* frames.

/ The next atom on the line (which should be a number) specify that the *previous* atom should be searched for that many times. For example, "@ FOO / 3" is equivalent to "@ FOO FOO FOO".

= Resets LASTPOS to the *value* of the next expression, e.g., if the value of FOO is a stack pointer, "@ = FOO FIE" will search for FIE in the environment specified by (the value of) FOO.

For example, if the push-down stack looks like:

BREAK1	*[9]*
FOO	*[8]*
COND	*[7]*
FIE	*[6]*
COND	*[5]*
FIE	*[4]*
COND	*[3]*
FIE	*[2]*
FUM	*[1]*

then "@ FIE COND" will set LASTPOS to the position corresponding to *[5]*; "@ @ COND" will then set LASTPOS to *[3]*; and "@ FIE / 3 -1" to *[1]*.

If @ cannot successfully complete a search for function *FN*, it searches the stack again from that point looking for a call to a function whose name is close to that of *FN*, in the sense of the spelling corrector (page 15.13). If the search is still unsuccessful, @ types (*FN* NOT FOUND), and then aborts.

When @ finishes, it types the name of the function at LASTPOS, i.e., (STKNAME LASTPOS).

@ can be used on BRKCOMS (see page 9.12). In this case, the *next* command on BRKCOMS is treated the same as the rest of the teletype line.

[3] When control passes from BREAK1, e.g. as a result of an EVAL, OK, GO, REVERT, ↑ command, or via a RETFROM or RETEVAL typed in by the user, (RELSTK LASTPOS) is executed to release this stack pointer.

?= [Break Command]

This is a multi-purpose command.[4] Its most common use is to interrogate the value(s) of the arguments of the broken function. For example, if FOO has three arguments (X Y Z), then typing ?= to a break on FOO will produce:

```
:?=
X = value of X
Y = value of Y
Z = value of Z
:
```

?= operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it operates on all of the arguments of the broken function. If the user types ?= X (CAR Y), he will see the value of X, and the value of (CAR Y).[5] The difference between using ?= and typing X and (CAR Y) directly to BREAK1 is that ?= evaluates its inputs as of the stack frame LASTPOS, i.e., it uses STKEVAL. This provides a way of examing variables or performing computations *as of a particular point on the stack*. For example, @ FOO / 2 followed by ?= X will allow the user to examine the value of X in the previous call to FOO, etc.

?= also recognizes numbers as referring to the correspondingly numbered argument, i.e., it uses STKARG in this case. Thus

```
:@ FIE
FIE
:?= 2
```

will print the name and value of the second argument of FIE.

?= can also be used on BRKCOMS (page 9.12, in which case the next command on BRKCOMS is treated as the rest of the teletype line. For example, if BRKCOMS is (EVAL ?= (X Y) GO), BRKEXP will be evaluated, the values of X and Y printed, and then the function exited with its value being printed.

PB [Break Command]

Prints the bindings of a given variable. Similar to ?=, except ascends the stack starting from LASTPOS, and, for each frame in which the given variable is bound, prints the frame name and value of the variable (with PRINTLEVEL reset to (2 . 3)), e.g.

```
:PB FOO
@  FN1:  3
@  FN2:  10
@  TOP:  NOBIND
```

[4] In fact, ?= is a universal mnemonic for displaying argument names and their corresponding values. In addition to being a break command, ?= is an edit macro which prints the argument names and values for the current expression (page 17.37), and a read-macro (actually ? is the read-macro character) which does the same for the current level list being read.

[5] The value of each variable is printed with the function SHOWPRINT (page 6.17), so that if SYSPRETTYFLG=T, the value will be prettyprinted.

PB is also a programmer's assistant command (page 8.14) that can be used when not in a break. PB is implemented via the function PRINTBINDINGS.

BT [Break Command]

Prints a backtrace of function names only starting at LASTPOS. The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries **BREAK**, **EDITOR**, and **TOP** respectively.

BTV [Break Command]

Prints a backtrace of function names *with* variables beginning at LASTPOS.

The value of each variable is printed with the function SHOWPRINT (page 6.17), so that if SYSPRETTYFLG=T, the value will be prettyprinted.

BTV+ [Break Command]

Same as BTV except also prints local variables and arguments to SUBRs.

BTV* [Break Command]

Same as BTV except prints arguments to SUBRs, local variables, and temporaries of the interpreter, i.e. eval blips (see page 7.10).

BTV! [Break Command]

Same as BTV except prints *everything* on the stack.

BT, BTV, BTV+, BTV*, and BTV! all take optional functional arguments. These arguments are used to choose functions to be *skipped* on the backtrace. As the backtrace scans down the stack, the name of each stack frame is passed to each of the functional arguments to the backtrace command. If any of these functions returns a non-NIL value, then that frame is skipped, and not shown in the backtrace. For example, BT SUBRP will skip all SUBRs, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used on BRKCOMS (page 9.12) the functional argument is no longer optional, i.e., the next element on BRKCOMS must either be a list of functional arguments, or NIL if no functional argument is to be applied.

For BT, BTV, BTV+, BTV*, and BTV!, if control-P is used to change a printlevel during the backtrace, the printlevel will be restored after the backtrace is completed.

The value of BREAKDELIMITER, initially "^{cr}", is printed to delimit the output of ?= and backtrace commands. This can be reset (e.g. to ",") for more linear output.

ARGS [Break Command]

Prints the names of the variables bound at LASTPOS, i.e., (VARIABLES LASTPOS) (page 7.5). For most cases, these are the arguments to the function entered at that position, i.e., (ARGLIST (STKNAME LASTPOS)).

REVERT [Break Command]

Goes back to position LASTPOS on stack and reenters the function called at that point with the arguments found on the stack. If the function is not already broken, REVERT first breaks it, and then unbreaks it after it is reentered.

REVERT can be given the position using the conventions described for @, e.g., REVERT FOO -1 is equivalent to @ FOO -1 followed by REVERT.

REVERT is useful for restarting a computation in the situation where a bug is discovered at some point *below* where the problem actually occurred. REVERT essentially says "go back there and start over in a break." REVERT will work correctly if the names or arguments to the function, or even its function type, have been changed.

ORIGINAL [Break Command]

For use in conjunction with BREAKMACROS (see page 9.12). Form is (ORIGINAL . *COMS*). *COMS* are executed without regard for BREAKMACROS. Useful for redefining a break command in terms of itself.

The following two commands are for use only with unbound atoms or undefined function breaks.

= *FORM* [Break Command]

Can only be used in a break following an unbound atom error. Sets the atom to the value of *FORM*, exits from the break returning that value, and continues the computation, e.g.,

```
UNBOUND ATOM

(FOO BROKEN)
:= (COPY FIE)
```

sets FOO and goes on.

Note: *FORM* may be given in the form *FN*[*ARGS*].

-> *EXPR* [Break Command]

Can be used in a break following either with unbound atom error, or an undefined function error. Replaces the expression containing the error with *EXPR* (not the value of *EXPR*), and continues the computation. -> does not just change BRKEXP; it changes the function or expression containing the erroneous form. In other words, the user does not have to perform any additional editing.

For example,

```
UNDEFINED CAR OF FORM

(FOO1 BROKEN)
:-> FOO
```

changes the FOO1 to FOO and continues the computation. *EXPR* need not be atomic, e.g.,

```
UNBOUND ATOM

(FOO BROKEN)
:-> (QUOTE FOO)
```

For undefined function breaks, the user can specify a function *and* initial arguments, e.g.,

```
UNDEFINED CAR OF FORM
```

```
(MEMBERX BROKEN)
:-> MEMBER X
```

Note that in the case of a undefined function error occurring immediately following a call to APPLY (e.g., (APPLY X Y) where the value of X is FOO and FOO is undefined), or a unbound atom error immediately following a call to EVAL (e.g., (EVAL X), where the value of X is FOO and FOO is unbound), there *is* no expression containing the offending atom. In this case, -> cannot operate, so ? is printed and no action is taken.

EDIT [Break Command]

Designed for use in conjunction with breaks caused by errors. Facilitates editing the expression causing the break:

```
NON-NUMERIC ARG
NIL
(IPLUS BROKEN)
:EDIT
IN FOO...
(IPLUS X Z)
EDIT
*(3 Y)
*OK
FOO
:
```

and the user can continue by typing OK, EVAL, etc.

This command is very simple conceptually, but complicated in its implementation by all of the exceptional cases involving interactions with compiled functions, breaks on user functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following simplified explanation which will account for 90% of the situations arising in actual usage. For those others, EDIT will print an appropriate failure message and return to the break.

EDIT begins by searching up the stack beginning at LASTPOS (set by @ command, initially position of the break) looking for a form, i.e., an internal call to EVAL. Then EDIT continues from that point looking for a call to an interpreted function, or to EVAL. It then calls the editor on either the EXPR or the argument to EVAL in such a way as to look for an expression EQ to the form that it first found. It then prints the form, and permits interactive editing to begin. Note that the user can then type successive 0's to the editor to see the chain of superforms for this computation.

If the user exits from the edit with an OK, the break expression is reset, if possible, so that the user can continue with the computation by simply typing OK. (Note that evaluating the new BRKEXP will involve reevaluating the form that causes the break, so that if (PUTD (QUOTE (FOO)) *BIG-COMPUTATION*) were handled by EDIT, *BIG-COMPUTATION* would be reevaluated.) However, in some situations, the break expression cannot be reset. For example, if a compiled function FOO incorrectly called PUTD and caused the error ARG NOT ATOM followed by a break on PUTD, EDIT might be able to find the form headed by FOO, and also find *that* form in some higher interpreted function. But after the user corrected the problem in the FOO-form, if any, he would still not have in any way informed EDIT what to do about the immediate problem, i.e., the incorrect call to PUTD. However, if FOO were *interpreted* EDIT would find the PUTD form itself, so that when the user corrected that form, EDIT could use the new corrected form to reset the break expression. The two cases are shown below:

If FOO is compiled:

```
FOO compiled                       FOO interpreted

ARG NOT ATOM                       ARG NOT ATOM
(FUM)                              (PUTD BROKEN)
(PUTD BROKEN)                      :EDIT
:EDIT                              IN FOO...
IN FIE...                          (PUTD X)
(FOO X)                            EDIT
EDIT                               *(2 (CAR X))
*(2 (CAR X))                       *OK
*OK                                :OK
NOTE: BRKEXP NOT CHANGED           PUTD
FIE
:?=
U = (FUM)
:(SETQ U (CAR U))
FUM
:OK
PUTD
```

IN? [Break Command]

Similar to EDIT, but just prints parent form, and superform, but does not call editor, e.g.,

```
ATTEMPT TO RPLAC NIL
T
(RPLACD BROKEN)
:IN?
FOO:  (RPLACD X Z)
```

Although EDIT and IN? were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function FOO, the user determines that there is a problem in the *call* to FOO, he can edit the calling form and reset the break expression with one operation by using EDIT. The following two protocol's with and without the use of EDIT, illustrate this:

```
Without EDIT:                      With EDIT:

(FOO BROKEN)                       (FOO BROKEN)
:?=                                :?=
X = (A B C)                        X = (A B C)
Y = D                              Y = D
:BT                                :EDIT
                                   *(SW 2 3)
FOO                                *OK
SETQ                               FIE[6]
COND                               :OK
PROG                               FOO
FIE
```

```
COND               find which function
                   FOO is called from
                   (aborted with ↑E)
:EDITF(FIE)
EDIT
*F FOO P
(FOO V U)          edit it
*(SW 2 3)
*OK
FIE
:(SETQ Y X)        reset X and Y
(A B C)
:(SETQQ X D)
D
:?=
X = D
Y = (A B C)        check them
:OK
FOO
```

9.2 WHEN TO BREAK

When an error occurs, the system has to decide whether to reset and unwind the stack, or go into a break. In the middle of a complex computation, it is usually helpful to go into a break, so that the user may examine the state of the computation. However, if the computation has only proceeded a little when the error occurs, such as when the user mistypes a function name, the user would normally just terminate a break, and it would be more convenient for the system to simply cause an error and unwind the stack in this situatuation. The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below; suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

(BREAKCHECK *ERRORPOS ERXN*) [Function]

BREAKCHECK is called by the error routine to decide whether or not to induce a break when a error occurs. *ERRORPOS* is the stack position at which the error occurred; *ERXN* is the error number. Returns T if a break should occur; NIL otherwise.

BREAKCHECK returns T (and a break occurs) if the "computation depth" is greater than or equal to HELPDEPTH. HELPDEPTH is initially set to 7, arrived at empirically by taking into account the overhead due to LISPX or BREAK.

If the depth of the computation is less than HELPDEPTH, BREAKCHECK next calculates the length of time spent in the computation. If this time is greater than

[6] X and Y have not been changed, but BRKEXP has.

HELPTIME milliseconds, initially set to 1000, then BREAKCHECK returns T (and a break occurs), otherwise NIL.

BREAKCHECK determines the "computation depth" by searching back up the stack looking for an ERRORSET frame (ERRORSETs indicate how far back unwinding is to take place when an error occurs, see page 9.15). At the same time, it counts the number of internal calls to EVAL. As soon as (if) the number of calls to EVAL exceeds HELPDEPTH, BREAKCHECK immediately stops searching for an ERRORSET and returns T. Otherwise, BREAKCHECK continues searching until either an ERRORSET is found or the top of the stack is reached. (Note: If the second argument to ERRORSET is INTERNAL, the ERRORSET is ignored by BREAKCHECK during this search.) BREAKCHECK then counts the number of function calls between the error and the last ERRORSET, or the top of the stack. The number of function calls plus the number of calls to EVAL (already counted) is used as the "computation depth".

BREAKCHECK determines the computation time by subtracting the value of the variable HELPCLOCK from the value of (CLOCK 2), the number of milliseconds of compute time (see page 14.10). HELPCLOCK is rebound to the current value of (CLOCK 2) for each computation typed in to LISPX or to a break. The time criterion for breaking can be suppressed by setting HELPTIME to NIL (or a very big number), or by setting HELPCLOCK to NIL. Note that setting HELPCLOCK to NIL will not have any effect beyond the current computation, because HELPCLOCK is rebound for each computation typed in to LISPX and BREAK.

The user can suppress all error breaks by setting the top level binding of the variable HELPFLAG to NIL using SETTOPVAL (HELPFLAG is bound as a local variable in LISPX, and reset to the global value of HELPFLAG on every LISPX line, so just SETQing it will not work.) If HELPFLAG=T (the initial value), the decision whether to cause an error or break is decided based on the computation time and the computation depth, as described above. Finally, if HELPFLAG=BREAK!, a break will always occur following an error.

9.3 BREAK1

The basic function of the break package is BREAK1, which creates a break. A break appears to be a regular executive, with the prompt ":", but BREAK1 also detects and interpretes break commands (page 9.3).

(BREAK1 *BRKEXP BRKWHEN BRKFN BRKCOMS BRKTYPE ERRORN*) [NLambda Function]

If *BRKWHEN* is NIL, *BRKEXP* is evaluated and returned as the value of BREAK1. Otherwise a break occurs and commands are then taken from *BRKCOMS* or the terminal and interpreted. All inputs not recognized by BREAK1 are simply passed on to the programmer's assistant.

When a break occurs, if *ERRORN* is a list whose CAR is a number, ERRORMESS is called to print an identifying message. If *ERRORN* is a list whose CAR is not a number, ERRORMESS1 is called. Otherwise, no preliminary message is printed. Following this, the message (*BRKFN* broken) is printed.

Since BREAK1 itself calls functions, when one of these is broken, an infinite loop would occur. BREAK1 detects this situation, and prints Break within a break on *FN*, and then simply calls the function without going into a break.

The commands GO, !GO, OK, !OK, RETURN and ↑ are the only ways to leave BREAK1. The command EVAL causes *BRKEXP* to be evaluated, and saves the value on the variable !VALUE. Other commands can be defined for BREAK1 via BREAKMACROS (below).

BRKTYPE is NIL for user breaks, INTERRUPT for control-H breaks, and ERRORX for error breaks. For breaks when *BRKTYPE* is not NIL, BREAK1 will clear and save the input buffer. If the break returns a value (i.e., is not aborted via ↑ or control-D) the input buffer will be restored.

The fourth argument to BREAK1 is *BRKCOMS*, a list of break commands that BREAK1 interprets and executes as though they were keyboard input. One can think of *BRKCOMS* as another input file which always has priority over the keyboard. Whenever *BRKCOMS*=NIL, BREAK1 reads its next command from the keyboard. Whenever *BRKCOMS* is not NIL, BREAK1 takes (CAR BRKCOMS) as its next command and sets *BRKCOMS* to (CDR BRKCOMS). For example, suppose the user wished to see the value of the variable X *after* a function was evaluated. He could set up a break with *BRKCOMS*=(EVAL (PRINT X) OK), which would have the desired effect. Note that if *BRKCOMS* is not NIL, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example. The function TRACE (page 10.4) uses *BRKCOMS*: it sets up a break with two commands; the first one prints the arguments of the function, or whatever the user specifies, and the second is the command GO, which causes the function to be evaluated and its value printed.

Note: If an error occurs while interpreting the *BRKCOMS* commands, *BRKCOMS* is set to NIL, and a full interactive break occurs.

The break package has a facility for redirecting ouput to a file. All output resulting from *BRKCOMS* will be output to the value of the variable BRKFILE, which should be the name of an open file. Output due to user typein is not affected, and will always go to the terminal. BRKFILE is initially T.

BREAKMACROS [Variable]

BREAKMACROS is a list of the form ((*NAME*$_1$ *COM*$_{11}$ ··· *COM*$_{1n}$) (*NAME*$_2$ *COM*$_{21}$ ··· *COM*$_{2n}$) ···). Whenever an atomic command is given to BREAK1, it first searches the list BREAKMACROS for the command. If the command is equal to *NAME*$_i$, BREAK1 simply appends the corresponding commands to the front of *BRKCOMS*, and goes on. If the command is not found on BREAKMACROS, BREAK1 then checks to see if it is one of the built in commands, and finally, treats it as a function or variable as before.[7]

Example: The command ARGS could be defined by including on BREAKMACROS the form: (ARGS (PRINT (VARIABLES LASTPOS T)))

(BREAKREAD *TYPE*) [Function]

Useful within BREAKMACROS for reading arguments. If BRKCOMS is non-NIL (the command in which the call to BREAKREAD appears was not typed in), returns the next break command from BRKCOMS, and sets BRKCOMS to (CDR BRKCOMS).

[7] If the command is not the name of a defined function, bound variable, or LISPX command, BREAK1 will attempt spelling correction using BREAKCOMSLST as a spelling list. If spelling correction is unsuccessful, BREAK1 will go ahead and call LISPX anyway, since the atom may also be a misspelled history command.

If BRKCOMS is NIL (the command was typed in), then BREAKREAD returns either the rest of the commands on the line as a list (if *TYPE*=LINE) or just the next command on the line (if *TYPE* is not LINE).

For example, the BT command is defined as (BAKTRACE LASTPOS NIL (BREAKREAD 'LINE) 0 T). Thus, if the user types BT, the third argument to BAKTRACE will be NIL. If the user types BT SUBRP, the third argument will be (SUBRP).

BREAKRESETFORMS [Variable]

If the user is developing programs that change the way a user and Interlisp normally interact (e.g., change or disable the interrupt or line-editing characters, turn off echoing, etc.), debugging them by breaking or tracing may be difficult, because Interlisp might be in a "funny" state at the time of the break. BREAKRESETFORMS is designed to solve this problem. The user puts on BREAKRESETFORMS expressions suitable for use in conjunction with RESETFORM or RESETSAVE (page 9.19). When a break occurs, BREAK1 evaluates each expression on BREAKRESETFORMS *before* any interaction with the terminal, and saves the values. When the break expression is evaluated via an EVAL, OK, or GO, BREAK1 first restores the state of the system with respect to the various expressions on BREAKRESETFORMS. When (if) control returns to BREAK1, the expressions on BREAKRESETFORMS are *again* evaluated, and their values saved. When the break is exited with an OK, GO, RETURN, or ↑ command, by typing control-D, or by a RETFROM or RETEVAL typed in by the user,[8] BREAK1 again restores state. Thus the net effect is to make the break invisible with respect to the user's programs, but nevertheless allow the user to interact in the break in the normal fashion.

As mentioned earlier, BREAK1 detects "Break within a break" situations, and avoids infinite loops. If the loop occurs because of an error, BREAK1 simply rebinds BREAKRESETFORMS to NIL, and calls HELP. This situation most frequently occurs when there is a bug in a function called by BREAKRESETFORMS.

Note: SETQ expressions can also be included on BREAKRESETFORMS for saving and restoring system parameters, e.g. (SETQ LISPXHISTORY NIL), (SETQ DWIMFLG NIL), etc. These are handled specially by BREAK1 in that the current value of the variable is saved before the SETQ is executed, and upon restoration, the variable is set back to this value.

9.4 ERROR FUNCTIONS

(ERRORX *ERXM*) [Function]

The entry to the error routines. If *ERXM*=NIL, (ERRORN) is used to determine the error-message. Otherwise, (SETERRORN (CAR *ERXM*) (CADR *ERXM*)) is performed, "setting" the error number and argument. Thus following either

[8] All user type-in is scanned in order to make the operations undoable as described on page 8.22. At this point, RETFROMs and RETEVALs are also noticed. However, if the user types in an expression which calls a function that then does a RETFROM, this RETFROM will not be noticed, and the effects of BREAKRESETFORMS will *not* be reversed.

(ERRORX '(10 T)) or (PLUS T), (ERRORN) is (10 T). ERRORX calls BREAKCHECK, and either induces a break or prints the message and unwinds to the last ERRORSET (page 9.10). Note that ERRORX can be called by any program to intentionally induce an error of any type. However, for most applications, the function ERROR will be more useful.

(ERROR *MESS1 MESS2 NOBREAK*) [Function]

Prints *MESS1* (using PRIN1), followed by a space if *MESS1* is an atom, otherwise a carriage return. Then *MESS2* is printed (using PRIN1 if *MESS2* is a string, otherwise PRINT). For example, (ERROR "NON-NUMERIC ARG" T) prints

```
NON-NUMERIC ARG
T
```

and (ERROR 'FOO "NOT A FUNCTION") prints FOO NOT A FUNCTION. If both *MESS1* and *MESS2* are NIL, the message printed is simply ERROR.

If *NOBREAK*=T, ERROR prints its message and then calls ERROR!.[9] Otherwise it calls (ERRORX '(17 (*MESS1* . *MESS2*))), i.e., generates error number 17, in which case the decision as to whether or not to break, and whether or not to print a message, is handled as per any other error.

(HELP *MESS1 MESS2 BRKTYPE*) [Function]

Prints *MESS1* and *MESS2* similar to ERROR, and then calls BREAK1 passing *BRKTYPE* as the BRKTYPE argument. If both *MESS1* and *MESS2* are NIL, HELP! is used for the message. HELP is a convenient way to program a default condition, or to terminate some portion of a program which the computation is theoretically never supposed to reach.

(SHOULDNT *MESS*) [Function]

Useful in those situations when a program detects a condition that should never occur. Calls HELP with the message arguments *MESS* and "Shouldn't happen!" and a BRKTYPE argument of 'ERRORX.

(ERROR!) [Function]

Programmable control-E; immediately returns from last ERRORSET or resets.

(RESET) [Function]

Programmable control-D; immediately returns to the top level.

(ERRORN) [Function]

Returns information about the last error in the form (*NUM EXP*) where *NUM* is the error number (page 9.22) and *EXP* is the expression which was (would have been) printed out after the error message. For example, following (PLUS T), (ERRORN) would return (10 T).

(SETERRORN *NUM MESS*) [Function]

Sets the value returned by ERRORN to (*NUM MESS*).

[9] unless the value of HELPFLAG is BREAK!, in which case a break will always occur (see page 9.11).

(ERRORMESS *U*) [Function]

Prints message corresponding to an ERRORN that yielded *U*. For example, (ERRORMESS '(10 T)) would print

```
NON-NUMERIC ARG
T
```

(ERRORMESS1 *MESS1 MESS2 MESS3*) [Function]

Prints the message corresponding to a HELP or ERROR break.

(ERRORSTRING *N*) [Function]

Returns as a new string the message corresponding to error number *N*, e.g., (ERRORSTRING 10)="NON-NUMERIC ARG".

(ERRORSET *FORM FLAG —*) [Function]

Performs (EVAL *FORM*). If no error occurs in the evaluation of *FORM*, the value of ERRORSET is a list containing one element, the value of (EVAL *FORM*). If an error did occur, the value of ERRORSET is NIL.

Note that ERRORSET is a lambda function, so its arguments are evaluated *before* it is entered, i.e., (ERRORSET X) means EVAL is called with the *value* of X. In most cases, ERSETQ and NLSETQ (described below) are more useful.

The argument *FLAG* controls the printing of error messages if an error occurs:

If *FLAG*=T, the error message is printed; if *FLAG*=NIL it is not (unless NLSETQGAG is NIL, see below). Note that if a *break* occurs below an ERRORSET, the message is printed regardless of the value of *FLAG*.

If *FLAG*=INTERNAL, this ERRORSET is ignored for the purpose of deciding whether or not to break or print a message (see page 9.10). However, the ERRORSET is in effect for the purpose of flow of control, i.e., if an error occurs, this ERRORSET returns NIL.

If *FLAG*=NOBREAK, no break will occur, even if the time criterion for breaking is met. Note that *FLAG*=NOBREAK will *not* prevent a break from occurring if the error occurs more than HELPDEPTH function calls below the errorset, since BREAKCHECK will stop searching before it reaches the ERRORSET. To guarantee that no break occurs, the user would also either have to reset HELPDEPTH or HELPFLAG.

(ERSETQ *FORM*) [NLambda Function]

Performs (ERRORSET '*FORM* T), evaluating *FORM* and printing error messages.

(NLSETQ *FORM*) [NLambda Function]

Performs (ERRORSET '*FORM* NIL), evaluating *FORM* without printing error messages.

NLSETQGAG [Variable]

If NLSETQGAG is NIL, error messages will print, regardless of the *FLAG* argument of ERRORSET. NLSETQGAG effectively changes all NLSETQs to ERSETQs. NLSETQGAG is initially T.

9.5 ERROR HANDLING BY ERROR TYPE

Occasionally the user may want to treat certain types of errors differently from others, e.g., always break, never break, or perhaps take some corrective action. This can be accomplished via ERRORTYPELST:

ERRORTYPELST [Variable]

ERRORTYPELST is a list of elements of the form (*NUM FORM*$_1$ ··· *FORM*$_N$), where *NUM* is one of the error numbers (page 9.22). During an error, after BREAKCHECK has been completed, but before any other action is taken, ERRORTYPELST is searched for an element with the same error number as that causing the error. If one is found, the corresponding forms are evaluated, and if the last one produces a non-NIL value, this value is substituted for the offender, and the function causing the error is reentered.

Within ERRORTYPELST entries, the following variables may be useful:

ERRORMESS [Variable]

CAR is the error number, CADR the "offender", e.g., (10 NIL) corresponds to a NON-NUMERIC ARG NIL error.

ERRORPOS [Variable]

Stack pointer to the function in which the error occurred, e.g., (STKNAME ERRORPOS) might be IPLUS, RPLACA, INFILE, etc.

Note: If the error is going to be handled by a RETFROM, RETTO, or a RETEVAL in the ERRORTYPELST entry, it probably is a good idea to first release the stack pointer ERRORPOS, e.g. by performing (RELSTK ERRORPOS).

BREAKCHK [Variable]

Value of BREAKCHECK, i.e., T means a break will occur, NIL means one will not. This may be reset within the ERRORTYPELST entry.

PRINTMSG [Variable]

If T, means print error message, if NIL, don't print error message, i.e., corresponds to second argument to ERRORSET. The user can force or suppress the printing of error message for various errortypes by including on ERRORTYPELST an expression which explicitly sets PRINTMSG.

For example, putting

```
[10 (AND (NULL (CADR ERRORMESS))
         (SELECTQ (STKNAME ERRORPOS)
                  ((IPLUS ADD1 SUB1) 0)
                  (ITIMES 1)
                  (PROGN (SETQ BREAKCHK T) NIL]
```

on ERRORTYPELST would specify that whenever a NON-NUMERIC ARG - NIL error occurred, and the function in question was IPLUS, ADD1, or SUB1, 0 should be used for the NIL. If the function was ITIMES, 1 should be used. Otherwise, always break. Note that the latter case is achieved not by the value returned, but by the *effect* of the evaluation, i.e., setting BREAKCHK to T. Similarly, (16 (SETQ BREAKCHK NIL)) would prevent END OF FILE errors from ever breaking.

ERRORTYPELST is initially ((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG))), which causes SPELLFILE to be called in case of a FILE NOT FOUND error (see page 15.20). If SPELLFILE is successful, the operation will be reexecuted with the new (corrected) file name.

9.6 INTERRUPT CHARACTERS

Errors and breaks can be caused by errors within functions, or by explicitly breaking a function. The user can also indicate his desire to go into a break at while a program is running by typing certain control characters known as "interrupt characters". The interrupt characters in Interlisp-D are listed on page 18.1; those in Interlisp-10 are listed on page 22.1.

The user can disable and/or redefine Interlisp interrupt characters, as well as define new interrupt characters. Interlisp-10 is initialized with 9 interrupt channels: RESET (control-D), ERROR (control-E), BREAK (control-B), HELP (control-H), PRINTLEVEL (control-P), CONTROL-T (control-T), RUBOUT (del), STORAGE (control-S), and OUTPUTBUFFER (control-O). Interlisp-D does not have the STORAGE and OUTPUTBUFFER interrupt channels, and has the additional channel RAID (control-C). Each of these channels independently can be disabled, or have a new interrupt character assigned to it via the function INTERRUPTCHAR described below. In addition, the user can enable up to 9 new interrupt channels, and associate with each channel an interrupt character and an expression to be evaluated when that character is typed.

User interrupts can be either "hard" or "soft". A "hard" interrupt is like control-E or control-D: it takes place as soon as it is typed. A soft interrupt is like control-H; it does not occur until the next function call. Soft interrupts can always be safely continued from. Hard interrupts rip the system out of the function currently being executed and unwind back to the last function call, i.e. part of the computation that was interrupted is lost and cannot be continued.

Hard interrupts are implemented by generating error number 43, and retrieving the corresponding form from the list USERINTERRUPTS once inside of ERRORX. Soft interrupts are implemented by calling INTERRUPT with an appropriate third argument, and then obtaining the corresponding form from USERINTERRUPTS. As soon as a soft interrupt character is typed, Interlisp clears and saves the input buffers, and then rings the bell. After the interrupt form is evaluated, the input buffers are restored. In either case, if a character is enabled as a user interrupt, but for some reason it is not found on USERINTERRUPTS, an UNDEFINED USER INTERRUPT error will be generated.

(INTERRUPTCHAR *CHAR* *TYP/FORM* *HARDFLG*) [Function]

Defines *CHAR* as an interrupt character. If *CHAR* was previously defined as an interrupt character, that interpretation is disabled.

CHAR is either a character or a character code (as returned by CHCON1). TENEX requires that interrupt characters be one of control-A, B,...,Z, space, esc(alt-mode), rubout(delete), or break.

If *TYP/FORM*=NIL, *CHAR* is disabled.

If *TYP/FORM*=T, the current state of *CHAR* is returned without changing or disabling it.

If *TYP/FORM* is one of the 8 literal atoms HELP, PRINTLEVEL, STORAGE, RUBOUT, ERROR, RESET, OUTPUTBUFFER, or BREAK, then INTERRUPTCHAR assigns *CHAR* to the indicated Interlisp interrupt channel, (reenabling the channel if previously disabled).

If *TYP/FORM* is any other literal atom, *CHAR* is enabled as an interrupt character that when typed causes the atom *TYP/FORM* to be *immediately* set to T.

If *TYP/FORM* is a list, *CHAR* is enabled as a user interrupt character, and *TYP/FORM* is the form that is evaluated when *CHAR* is typed. The interrupt will be hard if *HARDFLG*=T, otherwise soft.

(INTERRUPTCHAR T) restores all Interlisp channels to their original state, and disables all user interrupts.

INTERRUPTCHAR returns an expression which, when given as an argument to INTERRUPTCHAR, will restore things as they were before the call to INTERRUPTCHAR. Therefore, INTERRUPTCHAR can be used in conjunction with RESETFORM or RESETLST (page 9.20).

INTERRUPTCHAR is undoable.

(RESET.INTERRUPTS *PERMITTEDINTERRUPTS SAVECURRENT?*) [Function]

PERMITTEDINTERRUPTS is a list of interrupt character settings to be performed, each of the form (*CHAR* . *TYP/FORM*). The effect of RESET.INTERRUPTS is as if (INTERRUPTCHAR *CHAR TYP/FORM*) were performed for each item on *PERMITTEDINTERRUPTS*, and (INTERRUPTCHAR *OTHERCHAR* NIL) were performed on every other existing interrupt character.

If *SAVECURRENT?* is non-NIL, then RESET.INTERRUPTS returns the current state of the interrupts in a form that could be passed to RESET.INTERRUPTS, otherwise it returns NIL. This can be used with a RESET.INTERRUPTS that appears in a RESETFORM, so that the list is built at "entry", but not upon "exit".

(INTERRUPTABLE *FLAG*) [Function]

if *FLAG*=NIL, turns interrupt off. If *FLAG*=T, turns interrupt on. Value is previous setting. INTERRUPTABLE compiles open.

Note: Any interrupt character typed while interrupts are off is treated the same as any other character, i.e. placed in the input buffer, and will not cause an interrupt when interrupts are turned back on.

(INTERRUPTABLEP) [Function]

(Interlisp-10) Returns T if interrupts are enabled; NIL if disabled.

9.7 CHANGING AND RESTORING SYSTEM STATE

In Interlisp, a computation can be interrupted/aborted at any point due to an error, or more forcefully, because a control-D was typed, causing return to the top level. This situation creates problems for programs that need to perform a computation with the system in a "different state", e.g., different radix, input file, readtable, etc. but want to "protect" the calling environment, i.e., be able to restore the state when the computation has completed. While program errors and control-E can be "caught" by errorsets, control-D is not.[10] Thus the system may be left in its changed state as a result of the computation being aborted. The following functions address this problem.

Note that these functions do not and cannot handle the situation where their environment is exited via anything other than a normal return, an error, or a reset. E.g. a RETEVAL, RETFROM, RESUME, etc., will never be seen.

(RESETLST *FORM*$_1$ ··· *FORM*$_N$) [NLambda NoSpread Function]

RESETLST evaluates its arguments in order, after setting up an ERRORSET so that any reset operations performed by RESETSAVE (see below) are restored when the forms have been evaluated (or an error occurs, or a control-D is typed). If no error occurs, the value of RESETLST is the value of *FORM*$_N$, otherwise RESETLST generates an error (after performing the necessary restorations).

RESETLST compiles open.

(RESETSAVE *X* *Y*) [NLambda NoSpread Function]

RESETSAVE is used within a call to RESETLST to change the system state by calling a function or setting a variable, while specifying how to restore the original system state when the RESETLST is exited (normally, or with an error or control-D).

If *X* is atomic, resets the top level value of *X* to the value of *Y*. For example, (RESETSAVE LISPXHISTORY EDITHISTORY) resets the value of LISPXHISTORY to the value of EDITHISTORY, and provides for the original value of LISPXHISTORY to be restored when the RESETLST completes operation, (or an error occurs, or a control-D is typed). This use is somewhat anachronistic in Interlisp-10 in that in a shallow bound system, it is sufficient to simply rebind the variable. Furthermore, if there are any rebindings, the RESETSAVE will *not* affect the most recent binding but will change only the top level value, and therefore probably not have the intended effect.

If *X* is not atomic, it is a form that is evaluated. If *Y* is NIL, *X* must return as its value its "former state", so that the effect of evaluating the form can be reversed, and the system state can be restored, by applying CAR of *X* to the value of *X*. For example, (RESETSAVE (RADIX 8)) performs (RADIX 8), and provides for RADIX to be reset to its original value when the RESETLST completes by applying RADIX to the value returned by (RADIX 8).

In the special case that CAR of *X* is SETQ, the SETQ is transparent for the purposes of RESETSAVE, i.e. the user could also have written (RESETSAVE (SETQ X (RADIX 8))), and restoration would be performed by applying RADIX, not SETQ, to the previous value of RADIX.

If *Y* is not NIL, it is evaluated (before *X*), and its *value* is used as the restoring expression. This is useful for functions which do not return their "previous setting". For example,

[10] Note that the program could redefine control-D as a user interrupt (page 9.17), check for it, reenable it, and call RESET or something similar.

```
[RESETSAVE (SETBRK ···) (LIST 'SETBRK (GETBRK]
```

will restore the break characters by applying SETBRK to the value returned by (GETBRK), which was computed before the (SETBRK ···) expression was evaluated. Note that the restoration expression is still "evaluated" by *applying* its CAR to its CDR.

If *X* is NIL, *Y* is still treated as a restoration expression. Therefore,

```
(RESETSAVE NIL (LIST 'CLOSEF FILE))
```

will cause FILE to be closed when the RESETLST that the RESETSAVE is under completes (or an error occurs or a control-D is typed).

Note: RESETSAVE can be called when *not* under a RESETLST. In this case, the restoration will be performed at the next RESET, i.e., control-D or call to RESET. In other words, there is an "implicit" RESETLST at the top-level executive.

RESETSAVE compiles open. Its value is not a "useful" quantity.

(RESETVAR *VAR NEWVALUE FORM*) [NLambda Function]

Simplified form of RESETLST and RESETSAVE for resetting and restoring global variables.[11] Equivalent to (RESETLST (RESETSAVE *VAR NEWVALUE*) *FORM*). For example, (RESETVAR LISPXHISTORY EDITHISTORY (FOO)) resets LISPXHISTORY to the value of EDITHISTORY while evaluating (FOO). RESETVAR compiles open. If no error occurs, its value is the value of *FORM*.

(RESETVARS *VARSLST* E_1 E_2 ··· E_N) [NLambda NoSpread Function]

Similar to PROG, except the variables in *VARSLST* are global variables. In a shallow bound system (Interlisp-10) RESETVARS and PROG are identical.[12] In a deep bound system, each variable is "rebound" using RESETSAVE.

RESETVARS, like GETATOMVAL and SETATOMVAL (page 2.6), is provided to permit compatibility (i.e. transportablility) between a shallow bound and deep bound system with respect to conceptually global variables.

(RESETFORM *RESETFORM* $FORM_1$ $FORM_2$ ··· $FORM_N$) [NLambda NoSpread Function]

Simplified form of RESETLST and RESETSAVE for resetting a system state when the corresponding function returns as its value the "previous setting." Equivalent to (RESETLST (RESETSAVE *RESETFORM*) $FORM_1$ $FORM_2$ ··· $FORM_N$). For example, (RESETFORM (RADIX 8) (FOO)). RESETFORM compiles open. If no error occurs, it returns the value returned by $FORM_N$.

For some applications, the restoration operation must be different depending on whether the computation completed successfully or was aborted by an error or control-D. To facilitate this, while the restoration operation is being performed, the value of RESETSTATE will be bound to NIL, ERROR, or RESET,

[11] Unnecessarily expensive in a shallow bound system as the variable can simply be rebound.

[12] Except that the compiler insures that variables bound in a RESETVARS are declared as SPECVARS (see page 12.4).

depending on whether the exit was normal, due to an error, or reset (i.e., control-D, or in Interlisp-10, control-C followed by reenter). For example,

```
(RESETLST
   (RESETSAVE (INFILE X)
      (LIST '[LAMBDA (FL)
                   (COND ( (EQ RESETSTATE 'RESET)
                           (CLOSEF FL)
                           (DELFILE FL]
         X))
   FORMS)
```

will cause X to be closed and deleted only if a control-D was typed during the execution of *FORMS*.

When specifying complicated restoring expressions, it is often necessary to use the old value of the saving expression. For example, the following expression will set the primary input file (to FL) and execute some forms, but reset the primary input file only if an error or control-D occurs.

```
(RESETLST
   (SETQ TEM (INPUT FL))
   (RESETSAVE NIL
      (LIST '(LAMBDA (X) (AND RESETSTATE (INPUT X)))
            TEM))
   FORMS)
```

So that you will not have to explicitely save the old value, the variable OLDVALUE is bound at the time the restoring operation is performed to the value of the saving expression. Using this, the previous example could be recoded as:

```
(RESETLST
   (RESETSAVE (INPUT FL)
      '(AND RESETSTATE (INPUT OLDVALUE)))
   FORMS)
```

As mentioned earlier, restoring is performed by applying CAR of the restoring expression to the CDR, so RESETSTATE and (INPUT OLDVALUE) will not be evaluated by the APPLY. This particular example works because AND is an nlambda function that explicitly evaluates its arguments, so APPLYing AND to (RESETSTATE (INPUT OLDVALUE)) is the same as EVALing (AND RESETSTATE (INPUT OLDVALUE)). PROGN also has this property, so you can use a lambda function as a restoring form by enclosing it within a PROGN.

The function RESETUNDO (page 8.25) can be used in conjunction with RESETLST and RESETSAVE to provide a way of specifying that the system be restored to its prior state by *undoing* the side effects of the computations performed under the RESETLST.

9.8 ERROR LIST

There are currently fifty-plus types of errors in the Interlisp system. Some of these errors are implementation dependent, i.e., appear in Interlisp-10 but may not appear in other Interlisp systems.

Error List

The error number is set internally by the code that detects the error before it calls the error handling functions. It is also the value returned by ERRORN if called subsequent to that type of error, and is used by ERRORMESS for printing the error message.

Most errors will print the offending expression following the message, e.g., NON-NUMERIC ARG NIL is very common. Error number 18 (control-B) always causes a break (unless HELPFLAG is NIL). All other errors cause breaks if BREAKCHECK returns T (see page 9.10).

The errors are listed below by error number:

0 - JSYS ERROR (Interlisp-10) Occurs following a trap in a JSYS. As described on page 22.6, TRAP AT LOCATION is printed, followed by the JSYS diagnostic, and control returns to the operating system executive. The user can then safely CONTINUE, and the Interlisp error, JSYS ERROR is then generated. A TRAP AT LOCATION can also occur if an illegal instruction is executed. In this case, the operating system also prints ILLEGAL INSTRUCTION. This can happen for example if the user is programming directly in ASSEMBLE code, or if his system somehow got smashed. In the latter case, it is quite possible that random programs or data structures might have already been smashed. Unless he is sure he knows what the problem is, the user is best advised to abandon this system as soon as possible. (If the user does elect to CONTINUE, Interlisp will (try to) generate a JSYS ERROR and unwind. In some cases, however, the system may be so badly smashed that the error message won't even print.) Note that in some cases, e.g. illegal instruction trap while in the garbage collector, Interlisp will print out CAN'T CONTINUE, because traps under those conditions are fatal. The user *may* be able to reenter his sytem via the START command, and, if lucky, dump some data or functions before the system totally collapses.

In Interlisp-D, this error is named SYSTEM ERROR.

1 No longer used.

2 - STACK OVERFLOW

Occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. Usually because of a non-terminating recursive computation, i.e., a bug.

In Interlisp-10, the garbage collector uses the same stack as the rest of the system, so that if a garbage collection occurs when deep in a computation, the stack can overflow (particularly if there is a lot of list structure that is deep in the CAR direction). If this does happen, the garbage collector will flush the stack used by the computation in order that the garbage collection can complete. Afterwards, the error message STACK OVERFLOW IN GC - COMPUTATION LOST is printed, followed by a (RESET), i.e., return to top level.

3 - ILLEGAL RETURN

Call to RETURN when not inside of an interpreted PROG.

4 - ARG NOT LIST E.g., RPLACA called on a non-list.

5 - HARD DISK ERROR

(Interlisp-D) An error with the local disk drive.

6 - ATTEMPT TO SET NIL
Via SET or SETQ

7 - ATTEMPT TO RPLAC NIL
Attempt either to RPLACA or to RPLACD NIL with something other than NIL.

8 - UNDEFINED OR ILLEGAL GO
GO when not inside of a PROG, or GO to nonexistent label.

9 - FILE WON'T OPEN
From INFILE or OUTFILE, page 6.2.

10 - NON-NUMERIC ARG
A numeric function e.g., IPLUS, ITIMES, IGREATERP, expected a number.

11 - ATOM TOO LONG
Attempted to create a litatom (via PACK, or typing one in, or reading from a file) with too many characters. In Interlisp-D, the maximum number of characters in a litatom is 255. In Interlisp-10, the maximum is 127 characters.

12 - ATOM HASH TABLE FULL
No room for any more (new) atoms.

In Interlisp-10, the atom hash table will automatically expand by a specified number of pages each time it fills up until an upper limit of 32K atoms is reached.

13 - FILE NOT OPEN
From an I/O function, e.g., READ, PRINT, CLOSEF.

14 - ARG NOT LITATOM
E.g., SETQ, PUTPROP, GETTOPVAL, etc., given a non-atomic arg.

15 - TOO MANY FILES OPEN
≥30, excluding the terminal.

16 - END OF FILE From an input function, e.g., READ, READC, RATOM. After the error, the file will then be closed.

Note: The entries on ERRORTYPELST (page 9.16) are processed before the file is closed, so that the user can intercept and process this error via an entry on ERRORTYPELST, thereby preventing the file from being closed. It is also possible to use an ERRORTYPELST entry to return a character as the value of the call to ERRORX, and the program will continue, e.g. returning "]" may be used to complete a read operation.

17 - ERROR Call to ERROR (page 9.14).

18 - BREAK Control-B was typed.

19 - ILLEGAL STACK ARG
A stack function expected a stack position and was given something else. This might occur if the arguments to a stack function are reversed. Also occurs if user specified a stack position with a function name, and that function was not found

on the stack. See page 7.1.

20 - FAULT IN EVAL
Artifact of bootstrap. Never occurs after FAULTEVAL has been defined as described earlier.

21 - ARRAYS FULL System will first initiate a garbage collection of array space, and if no array space is reclaimed, will then generate this error.

22 - FILE SYSTEM RESOURCES EXCEEDED
(Interlisp-10) Includes no more disk space, disk quota exceeded, directory full, too many jfbs, job full.

23 - FILE NOT FOUND
File name does not correspond to a file in the corresponding directory. Can also occur if file name is ambiguous.

Interlisp is initialized with an entry on ERRORTYPELST (page 9.16) to call SPELLFILE for error 23. SPELLFILE will search alternate directories or perform spelling correction on the connected directory. If SPELLFILE fails, then the user will see this error.

24 - BAD SYSOUT FILE
Date does not agree with date of MAKESYS, or file is not a sysout file at all (see page 14.3).

25 - UNUSUAL CDR ARG LIST
A form ends in a non-list other than NIL, e.g., (CONS T . 3).

26 - HASH TABLE FULL
See hash array functions, page 2.35.

27 - ILLEGAL ARG Catch-all error. Currently used by PUTD, EVALA, ARG, FUNARG, ALLOCATE, RPLSTRING, etc.

28 - ARG NOT ARRAY
ELT or SETA given an argument that is not a pointer to the beginning of an array (see page 2.33).

29 - ILLEGAL OR IMPOSSIBLE BLOCK
(Interlisp-10) From GETBLK or RELBLK (see page 22.20).

30 - STACK PTR HAS BEEN RELEASED
A released stack pointer was supplied as a stack descriptor for a purpose other than as a stack pointer to be re-used (see page 7.1).

31 - STORAGE FULL
Following a garbage collection, if a sufficient amount of words has not been collected, and there is no un-allocated space left in the system, this error is generated.

32 - ATTEMPT TO USE ITEM OF INCORRECT TYPE
Before a field of a user data type is changed, the type of the item is first checked

to be sure that it is of the expected type. If not, this error is generated (see page 3.14).

33 - ILLEGAL DATA TYPE NUMBER
The argument is not a valid user data type number (see page 3.14).

34 - DATA TYPES FULL
All available user data types have been allocated. (see page 3.14).

35 - ATTEMPT TO BIND NIL OR T
In a PROG or LAMBDA expression.

36 - TOO MANY USER INTERRUPT CHARACTERS
Attempt to enable a user interrupt character when all 9 user channels are currently enabled (see page 9.17).

37 - READ-MACRO CONTEXT ERROR
(Interlisp-10) Occurs when a READ is executed from within a read-macro function and the next token is a) or a] (see page 6.36).

38 - ILLEGAL READTABLE
The argument was expected to be a valid readtable (see page 6.32).

39 - ILLEGAL TERMINAL TABLE
The argument was expected to be a valid terminal table (see page 6.40).

40 - SWAPBLOCK TOO BIG FOR BUFFER
(Interlisp-10) An attempt was made to swap in a function/array which is too large for the swapping buffer. See SETSBSIZE, page 22.26.

41 - PROTECTION VIOLATION
(Interlisp-10) Attempt to open a file that user does not have access to. Also reference to unassigned device.

42 - BAD FILE NAME
Illegal character in file specification, illegal syntax, e.g. in Interlisp-10, two ;'s etc.

43 - USER BREAK Error corresponding to "hard" user-interrupt character. See page 9.17.

44 - UNBOUND ATOM
Unbound atom error. When this occurs, a variable (atom) was used which had neither a stack binding (wasn't an argument to a function nor a PROG variable) nor a top level value. The "culprit" ((CADR ERRORMESS)) is the atom. Note that if DWIM corrects the error, no error occurs and the error number is not set. However, if an error is going to occur, whether or not it will cause a break, the error number will be set.

45 - UNDEFINED CAR OF FORM
Undefined function error. When is occurs, a form was evaluated whose function position (CAR) does not have a definition as a function. Culprit is the form.

46 - UNDEFINED FUNCTION
This error is generated if APPLY is given an undefined function. Culprit is (LIST

FN ARGS)

47 - CONTROL-E The user typed Control-E.

48 - FLOATING UNDERFLOW
(Interlisp-D) Underflow during floating-point operation.

49 - FLOATING OVERFLOW
(Interlisp-D) Overflow during floating-point operation.

50 - OVERFLOW (Interlisp-D) Overflow during integer operation.

51 - ARG NOT HARRAY
(Interlisp-D) Signaled by hash array operations when given an argument that is not a hash array. (In Interlisp-10, this still triggers error 28, ARG NOT ARRAY).

52 - TOO MANY ARGUMENTS
(Interlisp-D) Signaled when too many arguments are given to a lambda-spread, lambda-nospread, or nlambda-spread function.

In addition, many system functions, e.g., DEFINE, ARGLIST, ADVISE, LOG, EXPT, etc, also generate errors with appropriate messages by calling ERROR (see page 9.14) which causes error number 17.

CHAPTER 10

BREAKING, TRACING, AND ADVISING

It is frequently useful to be able to modify the behavior of a function without actually editing its definition. Interlisp provides several different facilities for doing this. By "breaking" a function, the user can cause breaks to occur at various times in the running of an incomplete program, so that the program state can be inspected. "Tracing" a function causes information to be printed every time the function is entered or exited. These are very useful debugging tools.

"Advising" is a facility for specifying longer-term function modifications. Even system functions can be changed through advising.

10.1 BREAKING FUNCTIONS AND DEBUGGING

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the Interlisp system, there are three facilities which allow the user to (temporarily) modify selected function definitions so that he can follow the flow of control in his programs, and obtain this debugging information. All three redefine functions in terms of a system function, BREAK1 (see page 9.11).

BREAK modifies the definition of a function *FN*, so that whenever *FN* is called and a break condition (defined by the user) is satisfied, a function break occurs. The user can then interrogate the state of the machine, perform any computation, and continue or return from the call.

TRACE modifies a definition of a function *FN* so that whenever *FN* is called, its arguments (or some other values specified by the user) are printed. When the value of *FN* is computed it is printed also. (TRACE is a special case of BREAK).

BREAKIN allows the user to insert a breakpoint *inside* an expression defining a function. When the breakpoint is reached and if a break condition (defined by the user) is satisfied, a temporary halt occurs and the user can again investigate the state of the computation.

The following two examples illustrate these facilities. In the first example, the user traces the function FACTORIAL. TRACE redefines FACTORIAL so that it print its arguments and value, and then goes on with the computation. When an error occurs on the fifth recursion, a full interactive break occurs. The situation is then the same as though the user had originally performed BREAK(FACTORIAL) instead of TRACE(FACTORIAL), and the user can evaluate various Interlisp forms and direct the course of the computation. In this case, the user examines the variable N, and instructs BREAK1 to return 1 as the value of this cell to FACTORIAL. The rest of the tracing proceeds without incident. The user would then presumably edit FACTORIAL to change L to 1.

```
←PP FACTORIAL

(FACTORIAL
  [LAMBDA (N)
     (COND
        ((ZEROP N
           L)
        (T (ITIMES N (FACTORIAL (SUB1 N])
FACTORIAL
←TRACE(FACTORIAL)
(FACTORIAL)
←FACTORIAL(4)

FACTORIAL:
N = 4

    FACTORIAL:
    N = 3

        FACTORIAL:
        N = 2

            FACTORIAL:
            N = 1

                FACTORIAL:
                N = 0

U.B.A.
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
                FACTORIAL = 1
            FACTORIAL = 1
        FACTORIAL = 2
    FACTORIAL = 6
FACTORIAL = 24
24
←
```

In the second example, the user has constructed a non-recursive definition of FACTORIAL. He uses BREAKIN to insert a call to BREAK1 just after the PROG label LOOP. This break is to occur only on the last two iterations, when N is less than 2. When the break occurs, the user tries to look at the value of N, but mistakenly types NN. The break is maintained, however, and no damage is done. After examining N and M the user allows the computation to continue by typing OK. A second break occurs after the next iteration, this time with N=0. When this break is released, the function FACTORIAL returns its value of 120.

```
←PP FACTORIAL
(FACTORIAL
  [LAMBDA (N)
```

```
     (PROG ((M 1))
        LOOP (COND
                ((ZEROP N)
                 (RETURN M)))
             (SETQ M (ITIMES M N))
             (SETQ N (SUB1 N))
             (GO LOOP])
FACTORIAL
←BREAKIN(FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL
←FACTORIAL(5)

((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)

((FACTORIAL) BROKEN)
:N
0
:OK
(FACTORIAL)
120
←
```

Note: BREAK and TRACE can also be used on CLISP words which appear as CAR of form, e.g. FETCH, REPLACE, IF, FOR, DO, etc., even though these are not implemented as functions. For conditional breaking, the user can refer to the entire expression via the variable EXP, e.g. BREAK ((FOR (MEMB 'UNTIL EXP))).

(BREAK0 *FN WHEN COMS* — —) [Function]

Sets up a break on the function *FN*; returns *FN*. If *FN* is not defined, returns (*FN* NOT DEFINED).

BREAK0 redefines *FN* as a call to BREAK1 (page 9.11), with an equivalent definition of *FN* as BRKEXP, and *WHEN*, *FN*, *COMS* as *BRKWHEN*, *BRKFN*, *BRKCOMS*. Puts a GENSYM defined with the original definition of *FN* on the property list of *FN* under the property BROKEN . Puts (BREAK0 *WHEN COMS*) on the property list of *FN* under the property BRKINFO (for use in conjunction with REBREAK). Adds *FN* to the front of the list BROKENFNS.

If *FN* is non-atomic and of the form (*FN1* IN *FN2*), BREAK0 breaks every call to *FN1* from within *FN2*. This is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to BREAKIN described below, but can be performed even when *FN2* is compiled or blockcompiled, whereas BREAKIN only works on interpreted functions. If *FN1* is not found in *FN2*, BREAKO returns the value (*FN1* NOT FOUND IN *FN2*).

BREAKO breaks one function *inside* another by first calling a function which changes the name of *FN1* wherever it appears inside of *FN2* to that of a new function, *FN1*-IN-*FN2*, which is initially given the same function definition as *FN1*. Then BREAKO proceeds to break on *FN1*-IN-*FN2* exactly as described above. In addition to breaking *FN1*-IN-*FN2* and adding *FN1*-IN-*FN2* to the list BROKENFNS, BREAKO adds *FN1* to the property value for the property NAMESCHANGED on the property list of *FN2* and puts (*FN2* . *FN1*) on the property list of *FN1*-IN-*FN2* under the property ALIAS. This will enable UNBREAK to recognize what changes have been made and restore the function *FN2* to its original state.

If *FN* is nonatomic and not of the above form, BREAKO is called for each member of *FN* using the same values for *WHEN*, *COMS*, and *FILE*. This distributivity permits the user to specify complicated break conditions on several functions. For example,

```
(BREAKO '(FOO1 ((PRINT PRIN1) IN (FOO2 FOO3)))
        '(NEQ X T)
        '(EVAL ?= (Y Z) OK) )
```

will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2 and PRIN1-IN-FOO3.

If *FN* is non-atomic, the value of BREAKO is a list of the functions broken.

(BREAK *x*) [NLambda NoSpread Function]

Nlambda nospread function. For each atomic argument, it performs (BREAKO *ATOM* T). For each list, it performs (APPLY 'BREAKO *LIST*). For example, (BREAK FOO1 (FOO2 (GREATERP N 5) (EVAL))) is equivalent to (BREAKO 'FOO1 T) and (BREAKO 'FOO2 '(GREATERP N 5) '(EVAL)).

(TRACE *x*) [NLambda NoSpread Function]

Nlambda nospread function. For each atomic argument, it performs (BREAKO *ATOM* T '(TRACE ?= NIL GO))[1]

For each list argument, CAR is the function to be traced, and CDR the forms the user wishes to see, i.e., TRACE performs:

```
(BREAKO (CAR LIST) T (LIST 'TRACE '?= (CDR LIST) 'GO))
```

For example, (TRACE FOO1 (FOO2 Y)) will cause both FOO1 and FOO2 to be traced. All the arguments of FOO1 will be printed; only the value of Y will be printed for FOO2. In the special case that the user wants to see *only* the value,

[1] The flag TRACE is checked for in BREAK1 and causes the message "*FUNCTION* :" to be printed instead of (*FUNCTION* BROKEN).

he can perform (TRACE (*FUNCTION*)). This sets up a break with commands (TRACE ?= (NIL) GO).

Note: the user can always call BREAK0 himself to obtain combination of options of BREAK1 not directly available with BREAK and TRACE. These two functions merely provide convenient ways of calling BREAK0, and will serve for most uses.

(BREAKIN *FN WHERE WHEN COMS*) [NLambda Function]

BREAKIN is an nlambda function. *WHEN* and *COMS* are similar to *WHEN* and *COMS* for BREAK0, except that if *WHEN* is NIL, T is used. *WHERE* specifies where in the definition of *FN* the call to BREAK1 is to be inserted (see below).

If *FN* is a compiled function, BREAKIN returns (*FN* UNBREAKABLE) as its value.

If *FN* is interpreted, BREAKIN types SEARCHING... while it calls the editor. If the location specified by *WHERE* is not found, BREAKIN types (NOT FOUND) and exits. If it is found, BREAKIN puts T under the property BROKEN-IN and (*WHERE WHEN COMS*) under the the property BRKINFO on the property list of *FN*, and adds *FN* to the front of the list BROKENFNS.

Multiple break points, can be inserted with a single call to BREAKIN by using a list of the form ((BEFORE ···) ··· (AROUND ···)) for *WHERE*. It is also possible to call BREAK or TRACE on a function which has been modified by BREAKIN, and conversely to BREAKIN a function which has been redefined by a call to BREAK or TRACE.

BREAKIN enables the user to insert a break, i.e., a call to BREAK1, at a specified location in an interpreted function. For example, if FOO calls FIE, inserting a break in FOO before the call to FIE is similar to breaking FIE. However, BREAKIN can be used to insert breaks before or after PROG labels, particular SETQ expressions, or even the evaluation of a variable. This is because BREAKIN operates by calling the editor and actually inserting a call to BREAK1 at a specified point *inside* of the function.

The user specifies where the break is to be inserted by a sequence of editor commands. These commands are preceded by BEFORE, AFTER, or AROUND, which BREAKIN uses to determine what to do once the editor has found the specified point, i.e., put the call to BREAK1 BEFORE that point, AFTER that point, or AROUND that point. For example, (BEFORE COND) will insert a break before the first occurrence of COND, (AFTER COND 2 1) will insert a break after the predicate in the first COND clause, (AFTER BF (SETQ X &)) after the *last* place X is set. Note that (BEFORE TTY:) or (AFTER TTY:) permit the user to type in commands to the editor, locate the correct point, and verify it for himself using the P command if he desires, and exit from the editor with OK.[2] BREAKIN then inserts the break BEFORE, AFTER, or AROUND that point.

For BREAKIN BEFORE or AFTER, the break expression is NIL, since the value of the break is irrelevant. For breakin AROUND, the break expression will be the indicated form. In this case, the user can use the EVAL command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if the user inserted a break after a COND predicate, e.g., (AFTER (EQUAL X Y)), he would be powerless to alter the flow of computation if the predicate were not true, since the break would

[2] A STOP command typed to TTY: produces the same effect as an unsuccessful edit command in the original specification, e.g., (BEFORE CONDD). In both cases, the editor aborts, and BREAKIN types (NOT FOUND).

not be reached. However, by breaking (AROUND (EQUAL X Y)), he can evaluate the break expression, i.e., (EQUAL X Y), look at its value, and return something else if he wished.

The message typed for a BREAKIN break, is ((*FN*) BROKEN), where *FN* is the name of the function inside of which the break was inserted. Any error, or typing control-E, will cause the full identifying message to be printed, e.g., (FOO BROKEN AFTER COND 2 1).

A special check is made to avoid inserting a break inside of an expression headed by any member of the list NOBREAKS, initialized to (GO QUOTE *), since this break would never be activated. For example, if (GO L) appears before the label L, BREAKIN (AFTER L) will not insert the break inside of the GO expression, but skip this occurrence of L and go on to the next L, in this case the label L. Similarly, for BEFORE or AFTER breaks, BREAKIN checks to make sure that the break is being inserted at a "safe" place. For example, if the user requests a break (AFTER X) in (PROG ··· (SETQ X &) ···), the break will actually be inserted AFTER (SETQ X &), and a message printed to this effect, e.g., BREAK INSERTED AFTER (SETQ X &).

(UNBREAK *x*) [NLambda NoSpread Function]

Nlambda nospread function. It takes an indefinite number of functions modified by BREAK, TRACE, or BREAKIN and restores them to their original state by calling UNBREAK0. Returns list of values of UNBREAK0.

(UNBREAK) will unbreak all functions on BROKENFNS, in reverse order. It first sets BRKINFOLST to NIL.

(UNBREAK T) unbreaks just the first function on BROKENFNS, i.e., the most recently broken function.

(UNBREAK0 *FN* —) [Function]

Restores *FN* to its original state. If *FN* was not broken, value is (NOT BROKEN) and no changes are made. If *FN* was modified by BREAKIN, UNBREAKIN is called to edit it back to its original state. If *FN* was created from (*FN1* IN *FN2*), (i.e., if it has a property ALIAS), the function in which *FN* appears is restored to its original state. All dummy functions that were created by the break are eliminated. Adds property value of BRKINFO to (front of) BRKINFOLST.

Note: (UNBREAK0 '(*FN1* IN *FN2*)) is allowed: UNBREAK0 will operate on (*FN1*-IN-*FN2*) instead.

(UNBREAKIN *FN*) [Function]

Performs the appropriate editing operations to eliminate all changes made by BREAKIN. *FN* may be either the name or definition of a function. Value is *FN*. UNBREAKIN is automatically called by UNBREAK if *FN* has property BROKEN-IN with value T on its property list.

(REBREAK *x*) [NLambda NoSpread Function]

Nlambda nospread function for rebreaking functions that were previously broken without having to respecify the break information. For each function on *x*, REBREAK searches BRKINFOLST for break(s) and performs the corresponding operation. Value is a list of values corresponding to calls to BREAK0 or BREAKIN. If no information is found for a particular function, returns (*FN* - NO BREAK

INFORMATION SAVED).

(REBREAK) rebreaks everything on BRKINFOLST, so (REBREAK) is the inverse of (UNBREAK).

(REBREAK T) rebreaks just the first break on BRKINFOLST, i.e., the function most recently unbroken.

(CHANGENAME *FN FROM TO*) [Function]

Changes all occurrences of *FROM* to *TO* in *FN*. *FN* may be compiled or blockcompiled. Value is *FN* if *FROM* was found, otherwise NIL. Does not perform any modifications of property lists. Note that *FROM* and *TO* do not have to be functions, e.g., they can be names of variables, or any other literals.

(VIRGINFN *FN FLG*) [Function]

The function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by PRETTYPRINT, DEFINE, and the compiler. If *FLG*=NIL, as for PRETTYPRINT, it does not modify the definition of *FN* in the process of producing a "clean" version of the definition; it works on a copy. If *FLG*=T, as for the compiler and DEFINE, it physically restores the function to its original state, and prints the changes it is making, e.g., FOO UNBROKEN, FOO UNADVISED, FOO NAMES RESTORED, etc. Returns the virgin function definition.

10.2 ADVISING

The operation of advising gives the user a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the *interface* between functions as opposed to modifying the function definition itself, as in editing. BREAK, TRACE, and BREAKDOWN, are examples of the use of this technique: they each modify user functions by placing relevant computations *between* the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows the user to treat functions, his or someone else's, as "black boxes," and to modify them without concern for their contents or details of operations. For example, the user could modify SYSOUT to set SYSDATE to the time and date of creation by (ADVISE 'SYSOUT '(SETQ SYSDATE (DATE))).

As with BREAK, advising works equally well on compiled and interpreted functions. Similarly, it is possible to effect a modification which only operates when a function is called from some other specified function, i.e., to modify the interface between two particular functions, instead of the interface between one function and the rest of the world. This latter feature is especially useful for changing the *internal* workings of a system function.

For example, suppose the user wanted TIME (page 14.14) to print the results of his measurements to the file FOO instead of the teletype. He could accomplish this by (ADVISE '((PRIN1 PRINT SPACES) IN TIME) 'BEFORE '(SETQQ U FOO))

Note that advising PRIN1, PRINT, or SPACES directly would have affected all calls to these very frequently used function, whereas advising ((PRIN1 PRINT SPACES) IN TIME) affects just those calls to PRIN1, PRINT, and SPACES from TIME.

Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable !VALUE, as with BREAK1. For example, suppose the user wanted to perform some computation following each SYSIN, e.g., check whether his files were up to date. He could then: (ADVISE 'SYSOUT 'AFTER '(COND ((LISTP !VALUE) --))).[3]

10.2.1 Implementation of Advising

After a function has been modified several times by ADVISE, it will look like:

```
(LAMBDA arguments
   (PROG (!VALUE)
      (SETQ !VALUE
         (PROG NIL
              advice1
                 .
                 .      advice before
                 .
              advicen
              (RETURN BODY)))
      advice1
         .
         .      advice after
         .
      advicem
      (RETURN !VALUE)))
```

where *BODY* is equivalent to the original definition.[4] Note that the structure of a function modified by ADVISE allows a piece of advice to bypass the original definition by using the function RETURN. For example, if (COND ((ATOM X) (RETURN Y))) were one of the pieces of advice BEFORE a function, and this function was entered with X atomic, Y would be returned as the value of the inner PROG, !VALUE would be set to Y, and control passed to the advice, if any, to be executed AFTER the function. If this same piece of advice appeared AFTER the function, Y would be returned as the value of the entire advised function.

The advice (COND ((ATOM X) (SETQ !VALUE Y))) AFTER the function would have a similar effect, but the rest of the advice AFTER the function would still be executed.

Note: Actually, ADVISE uses its own versions of PROG, SETQ, and RETURN, (called ADV-PROG, ADV-SETQ, and ADV-RETURN) in order to enable advising these functions.

[3] After the SYSIN, the system will be as it was when the SYSOUT was performed, hence the advice must be to SYSOUT, not SYSIN. See page 14.3 for complete discussion of SYSOUT.

[4] If *FN* was originally an EXPR, *BODY* is the body of the definition, otherwise a form using a GENSYM which is defined with the original definition.

10.2.2 Advise Functions

ADVISE is a function of four arguments: *FN*, *WHEN*, *WHERE*, and *WHAT*. *FN* is the function to be modified by advising, *WHAT* is the modification, or piece of advice. *WHEN* is either BEFORE, AFTER, or AROUND, and indicates whether the advice is to operate BEFORE, AFTER, or AROUND the body of the function definition. *WHERE* specifies exactly where in the list of advice the new advice is to be placed, e.g., FIRST, or (BEFORE PRINT) meaning before the advice containing PRINT, or (AFTER 3) meaning after the third piece of advice, or even (: TTY:). If *WHERE* is specified, ADVISE first checks to see if it is one of LAST, BOTTOM, END, FIRST, or TOP, and operates accordingly. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the corresponding location.

Both *WHEN* and *WHERE* are optional arguments, in the sense that they can be omitted in the call to ADVISE. In other words, ADVISE can be thought of as a function of two arguments (ADVISE *FN* *WHAT*), or a function of three arguments: (ADVISE *FN* *WHEN* *WHAT*), or a function of four arguments: (ADVISE *FN* *WHEN* *WHERE* *WHAT*). Note that the advice is always the *last* argument. If *WHEN*=NIL, BEFORE is used. If *WHERE*=NIL, LAST is used.

(ADVISE *FN* *WHEN* *WHERE* *WHAT*) [Function]

FN is the function to be advised, *WHEN*=BEFORE, AFTER, or AROUND, *WHERE* specifies where in the advice list the advice is to be inserted, and *WHAT* is the piece of advice.

If *FN* is of the form (*FN1* IN *FN2*), *FN1* is changed to *FN1*-IN-*FN2* throughout *FN2*, as with break, and then *FN1*-IN-*FN2* is used in place of *FN*. If *FN1* and/or *FN2* are lists, they are distributed as with BREAK0, page 10,3.

If *FN* is broken, it is unbroken before advising.

If *FN* is not defined, an error is generated, NOT A FUNCTION.

If *FN* is being advised for the first time, i.e., if (GETP *FN* 'ADVISED)=NIL, a GENSYM is generated and stored on the property list of *FN* under the property ADVISED, and the GENSYM is defined with the original definition of *FN*. An appropriate S-expression definition is then created for *FN*.[5] Finally, *FN* is added to the (front of) ADVISEDFNS, so that (UNADVISE T) always unadvises the last function advised (see page 10.10).

If *FN* has been advised before, it is moved to the front of ADVISEDFNS.

If *WHEN*=BEFORE or AFTER, the advice is inserted in *FN*'s definition either BEFORE or AFTER the original body of the function. Within that context, its position is determined by *WHERE*. If *WHERE*=LAST, BOTTOM, END, or NIL, the advice is added following all other advice, if any. If *WHERE*=FIRST or TOP, the advice is inserted as the first piece of advice. Otherwise, *WHERE* is treated as a command for the editor, similar to BREAKIN, e.g., (BEFORE 3), (AFTER PRINT).

[5] Using private versions of PROG, SETQ, and RETURN, so that these functions can also be advised.

If *WHEN*=AROUND, the body is substituted for * in the advice, and the result becomes the new body, e.g., (ADVISE 'FOO 'AROUND '(RESETFORM (OUTPUT T) *)). Note that if several pieces of AROUND advice are specified, earlier ones will be embedded inside later ones. The value of *WHERE* is ignored.

Finally (LIST *WHEN WHERE WHAT*) is added (by ADDPROP) to the value of property ADVICE on the property list of *FN*, so that a record of all the changes is available for subsequent use in readvising. Note that this property value is a list of the advice in order of calls to ADVISE, not necessarily in order of appearance of the advice in the definition of *FN*.

The value of ADVISE is *FN*.

If *FN* is non-atomic, every function in *FN* is advised with the same values (but copies) for *WHEN*, *WHERE*, and *WHAT*. In this case, ADVISE returns a list of individual functions.

Note: advised functions can be broken. However if a function is broken at the time it is advised, it is first unbroken. Similarly, advised functions can be edited, including their advice. UNADVISE will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

(UNADVISE *X*) [NLambda NoSpread Function]

An nlambda nospread like UNBREAK. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by ADVISE. UNADVISE saves on the list ADVINFOLST enough information to allow restoring a function to its advised state using READVISE. ADVINFOLST and READVISE thus correspond to BRKINFOLST and REBREAK. If a function contains the property READVICE, UNADVISE moves the current value of the property ADVICE to READVICE.

(UNADVISE) unadvises all functions on ADVISEDFNS in reverse order, so that the most recently advised function is unadvised last. It first sets ADVINFOLST to NIL.

(UNADVISE T) unadvises the first function of ADVISEDFNS, i.e., the most recently advised function.

(READVISE *X*) [NLambda NoSpread Function]

An nlambda nospread like REBREAK for restoring a function to its advised state without having to specify all the advise information. For each function on *X*, READVISE retrieves the advise information either from the property READVICE for that function, or from ADVINFOLST, and performs the corresponding advise operation(s). In addition it stores this information on the property READVICE if not already there. If no information is found for a particular function, value is (*FN* - NO ADVICE SAVED).

(READVISE) readvises everything on ADVINFOLST.

(READVISE T) readvises the first function on ADVINFOLST, i.e., the function most recently unadvised.

A difference between ADVISE, UNADVISE, and READVISE versus BREAK, UNBREAK, and REBREAK, is that if a function is not rebroken between successive (UNBREAK)'s, its break information is forgotten. However, once READVISE is called on a function, that function's advice is permanently saved on its property list (under READVICE); subsequent calls to UNADVISE will not remove it. In fact, calls to UNADVISE update the property READVICE with the current value of the property ADVICE, so that the sequence READVISE, ADVISE, UNADVISE causes the augmented advice to become permanent. Note that the sequence READVISE, ADVISE, READVISE removes the "intermediate advice" by restoring the function to its earlier state.

(ADVISEDUMP *X* *FLG*) [Function]

Used by PRETTYDEF when given a command of the form (ADVISE ···) or (ADVICE ···). If *FLG*=T, ADVISEDUMP writes both a DEFLIST and a READVISE (this corresponds to (ADVISE ···)). If *FLG*=NIL, only the DEFLIST is written (this corresponds to (ADVICE ···)). In either case, ADVISEDUMP copies the advise information to the property READVICE, thereby making it "permanent" as described above.

CHAPTER 11

FILE PACKAGE

Most implementations of Lisp treat symbolic files as unstructured text, much as they are treated in most conventional programming environments. Function definitions are edited with a character-oriented text editor, and then the changed definitions (or sometimes the entire file) is read or compiled to install those changes in the running memory image. Interlisp incorporates a different philosophy. A symbolic file is considered as a database of information about a group of data objects—function definitions, variable values, record declarations, etc. The text in a symbolic file is never edited directly. Definitions are edited only after their textual representations on files have been converted to data-structures that reside inside the Lisp address space. The programs for editing definitions inside Interlisp can therefore make use of the full set of data-manipulation capabilities that the environment already provides, and editing operations can be easily intermixed with the processes of evaluation and compilation.

Interlisp is thus a "resident" programming environment, and as such it provides facilities for moving definitions back and forth between memory and the external databases on symbolic files, and for doing the bookkeeping involved when definitions on many symbolic files with compiled counterparts are being manipulated. The file package provides those capabilities. It removes from the user the burden of keeping track of where things are and what things have changed. The file package also keeps track of which files have been modified and need to be updated and recompiled.

The file package is integrated into many other system packages. For example, if only the compiled version of a file is loaded and the user attempts to edit a function, the file package will attempt to load the source of that function from the appropriate symbolic file. In many cases, if a datum is needed by some program, the file package will automatically retrieve it from a file if it is not already in the user's working environment.

Some of the operations of the file package are rather complex. For example, the same function may appear in several different files, or the symbolic or compiled files may be in different directories, etc. Therefore, this chapter does not document how the file package works in each and every situation, but instead makes the deliberately vague statement that it does the "right" thing with respect to keeping track of what has been changed, and what file operations need to be performed in accordance with those changes.

For a simple illustration of what the file package does, suppose that the symbolic file FOO contains the functions FOO1 and FOO2, and that the file BAR contains the functions BAR1 and BAR2. These two files could be loaded into the environment with the function LOAD:

```
← (LOAD 'FOO)
FILE CREATED  4-MAR-83 09:26:55
FOOCOMS
{DSK}FOO.;1
← (LOAD 'BAR)
FILE CREATED  4-MAR-83 09:27:24
BARCOMS
{DSK}BAR.;1
```

Now, suppose that we change the definition of FOO2 with the editor, and we define two new functions, NEW1 and NEW2. At that point, the file package knows that the in-memory definition of FOO2 is no longer consistent with the definition in the file FOO, and that the new functions have been defined but have not yet been associated with a symbolic file and saved on permanent storage. The function FILES? summarizes this state of affairs and enters into an interactive dialog in which we can specify what files the new functions are to belong to.

```
← (FILES?)
FOO...to be dumped.
    plus the functions: NEW1,NEW2
want to say where the above go ? Yes
(functions)
NEW1  File name: BAR
NEW2  File name: ZAP
  new file ? Yes
NIL
```

The file package knows that the file FOO has been changed, and needs to be dumped back to permanent storage. This can be done with MAKEFILE.

```
←(MAKEFILE 'FOO)
{DSK}FOO.;2
```

Since we added NEW1 to the old file BAR and established a new file ZAP to contain NEW2, both BAR and ZAP now also need to be dumped. This is confirmed by a second call to FILES?:

```
← (FILES?)
BAR, ZAP...to be dumped.
FOO...to be listed.
FOO...to be compiled
NIL
```

We are also informed that the new version we made of FOO needs to be listed (sent to a printer) and that the functions on the file must be compiled.

Rather than doing several MAKEFILEs to dump the files BAR and ZAP, we can simply call CLEANUP. Without any further user interaction, this will dump any files whose definitions have been modified. CLEANUP will also send any unlisted files to the printer and recompile any files which need to be recompiled. CLEANUP is a useful function to use at the end of a debugging session. It will call FILES? if any new objects have been defined, so the user does not lose the opportunity to say explicitly where those belong. In effect, the function CLEANUP executes all the operations necessary to make the user's permanent files consistent with the definitions in his current core-image.

```
← (CLEANUP)
FOO...compiling {DSK}FOO.;2
    .
    .
    .
BAR...compiling {DSK}BAR.;2
    .
    .
    .
```

```
ZAP...compiling {DSK}ZAP.;1
    .
    .
    .
```

In addition to the definitions of functions, symbolic files in Interlisp can contain definitions of a variety of other types, e.g. variable values, property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file package uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a litatom), a definition of a given type (called the file package type). Note that the same name may have several definitions of different types. For example, a litatom may have both a function definition and a variable definition. The file package also keeps track of the files that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

Symbolic files on permanent storage devices are referred to by names that obey the naming conventions of those devices, usually including host, directory, and version fields. When such definition groups are noticed by the file package, they are assigned simple *root names* and these are used by all file package operations to refer to those groups of definitions. The root name for a group is computed from its full permanent storage name by applying the function ROOTFILENAME; this strips off the host, directory, version, etc., and returns just the simple name field of the file. For each file, the file package also has a data structure that describes what definitions it contains. This is known as the commands of the file, or its "filecoms". By convention, the filecoms of a file whose root name is *x* is stored as the value of the litatom *x*COMS. For example, the value of FOOCOMS is the filecoms for the file FOO. This variable can be directly manipulated, but the file package contains facilities such as FILES? which make constructing and updating filecoms easier, and in some cases automatic. See page 11.32.

The file package is able to maintain its databases of information because it is notified by various other routines in the system when events take place that may change that database. A file is "noticed" when it is loaded, or when a new file is stored (though there are ways to explicitly notice files without completely loading all their definitions). Once a file is noticed, the file package takes it into account when modifying filecoms, dumping files, etc. The file package also needs to know what typed definitions have been changed or what new definitions have been introduced, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file package operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file package. Also, *typed-in* assignment of variables or property values is noticed by the file package. (Note that modifications to variable or property values during the execution of a function body are not noticed.) In some cases the marking procedure can be subtle, e.g. if the user edits a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

All file package operations can be disabled with FILEPKGFLG.

FILEPKGFLG [Variable]

The file package can be disabled by setting FILEPKGFLG to NIL. This will turn off noticing files and marking changes. FILEPKGFLG is initially T.

The rest of this chapter goes into further detail about the file package. Functions for loading and storing symbolic files are presented first, followed by functions for adding and removing typed definitions from files, moving typed definitions from one file to another, determining which file a particular definition is stored in, and so on.

11.1 LOADING FILES

The functions below load information from symbolic files into the Interlisp environment. A symbolic file contains a sequence of Interlisp expressions that can be evaluated to establish specified typed definitions. The expressions on symbolic files are read using FILERDTBL as the readtable.

The loading functions all have an argument *LDFLG*. *LDFLG* affects the operation of DEFINE, DEFINEQ, RPAQ, RPAQ?, and RPAQQ. While a source file is being loaded, DFNFLG (page 5.9) is rebound to *LDFLG*. Thus, if *LDFLG*=NIL, and a function is redefined, a message is printed and the old definition saved. If *LDFLG*=T, the old definition is simply overwritten. If *LDFLG*=PROP, the functions are stored as "saved" definitions on the property lists under the property EXPR instead of being installed as the active definitions. If *LDFLG*=ALLPROP, not only function definitions but also variables set by RPAQQ, RPAQ, RPAQ? are stored on property lists (except when the variable has the value NOBIND, in which case they are set to the indicated value regardless of DFNFLG).

Another option is available for users who are loading systems for others to use and who wish to suppress the saving of information used to aid in development and debugging. If *LDFLG*=SYSLOAD, LOAD will: (1) Rebind DFNFLG to T, so old definitions are simply overwritten; (2) Rebind LISPXHIST to NIL, thereby making the LOAD not be undoable and eliminating the cost of saving undo information (See page 8.22); (3) Rebind ADDSPELLFLG to NIL, to suppress adding to spelling lists; (4) Rebind FILEPKGFLG to NIL, to prevent the file from being "noticed" by the file package; (5) Rebind BUILDMAPFLG to NIL, to prevent a file map from being constructed; (6) After the load has completed, set the filecoms variable and any filevars variables[1] to NOBIND; and (7) Add the file name to SYSFILES rather than FILELST.

Note: All functions that have *LDFLG* as an argument perform spelling correction using LOADOPTIONS as a spelling list when *LDFLG* is not a member of LOADOPTIONS. LOADOPTIONS is initially (NIL T PROP ALLPROP SYSLOAD).

(LOAD *FILE LDFLG PRINTFLG*) [Function]

Reads successive expressions from *FILE* (with FILERDTBL as readtable) and evaluates each as it is read, until it reads either NIL, or the single atom STOP. Note that LOAD can be used to load both symbolic and compiled files. Returns *FILE* (full name).

If *PRINTFLG*=T, LOAD prints the value of each expression; otherwise it does not.

(LOAD? *FILE LDFLG PRINTFLG*) [Function]

Similar to LOAD except that it does not load *FILE* if it has already been loaded, in which case it returns NIL.

Note: The test is whether the root name of *FILE* has a FILEDATES property (page 11.13).

[1] A filevars variable is any variable appearing in a file package command of the form (*FILECOM* * *VARIABLE*) (see page 11.30). Therefore, if the filecoms includes (FNS * FOOFNS), FOOFNS is set to NOBIND. If the user wants the value of such a variable to be retained, even when the file is loaded with *LDFLG*=SYSLOAD, then he should replace the variable with an equivalent, *non-atomic* expression, such as (FNS * (PROGN FOOFNS)).

(LOADFNS *FNS FILE LDFLG VARS*) [Function]

Permits selective loading of definitions. *FNS* is a list of function names, a single function name, or T, meaning to load all of the functions on the file. *FILE* can be either a compiled or symbolic file. If a compiled definition is loaded, so are all compiler-generated subfunctions. The interpretation of *LDFLG* is the same as for LOAD.

If *FILE*=NIL, LOADFNS will use WHEREIS (page 11.10) to determine where the first function in *FNS* resides, and load from that file. Note that the file must previously have been "noticed" (see page 11.12). If WHEREIS returns NIL, and the WHEREIS package (page 23.40) has been loaded, LOADFNS will use the WHEREIS data base to find the file containing *FN*.

VARS specifies which non-DEFINEQ expressions are to be loaded (i.e., evaluated): T means all, NIL means none, VARS means to evaluate all variable assignment expressions (beginning with RPAQ, RPAQQ, or RPAQ?, see page 11.37), and any other atom is the same as specifying a list containing that atom.

If *VARS* is a list, each element in *VARS* is "matched" against each non-DEFINEQ expression, and if any elements in *VARS* "match" successfully, the expression is evaluated. "Matching" is defined as follows: If an element of *VARS* is an atom, it matches an expression if it is EQ to either the CAR or the CADR of the expression. If an element of *VARS* is a list, it is treated as an edit pattern (page 17.13), and matched with the entire expression (using EDIT4E, page 17.57). For example, if *VARS* was (FOOCOMS DECLARE: (DEFLIST & (QUOTE MACRO))), this would cause (RPAQQ FOOCOMS ···), all DECLARE:s, and all DEFLISTs which set up MACROs to be read and evaluated.

If *VARS* is a list and (FNTYP *VARS*) is true (*VARS* is a function definition), then LOADFNS will invoke that function on every non-DEFINEQ expression being considered, applying it to two arguments, the first and second elements in the expression. If the function returns NIL, the expression will be skipped; if it returns a non-NIL litatom (e.g. T), the expression will be evaluated; and if it returns a list, this list is evaluated instead of the expression. Note: The file pointer is set to the very beginning of the expression before calling the *VARS* function definition, so it may read the entire expression if necessary. If the function returns a litatom, the file pointer is reset and the expression is READ or SKREAD. However, the file pointer is not reset when the function returns a list, so the function must leave it set immediately after the expression that it has presumably read.

LOADFNS returns a list of: (1) The names of the functions that were found; (2) A list of those functions not found (if any) headed by the litatom NOT-FOUND:; (3) All of the expressions that were evaluated; (4) A list of those members of *VARS* for which no corresponding expressions were found (if any), again headed by the litatom NOT-FOUND:. For example,

```
← (LOADFNS '(FOO FIE FUM) FILE NIL '(BAZ (DEFLIST &)))
(FOO FIE (NOT-FOUND: FUM) (RPAQ BAZ ···) (NOT-FOUND: (DEFLIST
&)))
```

(LOADVARS *VARS FILE LDFLG*) [Function]

Same as (LOADFNS NIL *FILE LDFLG VARS*).

(LOADFROM *FILE FNS LDFLG*) [Function]

Same as (LOADFNS *FNS FILE LDFLG* T).

Once the file package has noticed a file, the user can edit functions contained in the file without explicitly loading them. Similarly, those functions which have not been modified do not have to be loaded in order to write out an updated version of the file. Files are normally noticed (i.e., their contents become known to the file package; see page 11.12) when either the symbolic or compiled versions of the file are loaded. If the file is *not* going to be loaded completely, the preferred way to notice it is with LOADFROM. Note that the user can also load some functions at the same time by giving LOADFROM a second argument, but it is normally used simply to inform the file package about the existence and contents of a particular file.

(LOADBLOCK *FN FILE LDFLG*) [Function]

Calls LOADFNS on those functions contained in the block declaration containing *FN* (See page 12.14). LOADBLOCK is designed primarily for use with symbolic files, to load the EXPRs for a given block. It will not load a function which already has an in-core EXPR definition, and it will not load the block name, unless it is also one of the block functions.

(LOADCOMP *FILE LDFLG*) [Function]

Performs all operations on *FILE* associated with compilation, i.e. evaluates all expressions under a DECLARE: EVAL@COMPILE (see page 11.26), and "notices" the function and variable names by adding them to the lists NOFIXFNSLST and NOFIXVARSLST (see page 16.16).

Thus, if building a system composed of many files with compilation information scattered among them, all that is required to compile one file is to LOADCOMP the others.

(LOADCOMP? *FILE LDFLG*) [Function]

Similar to LOADCOMP, except it does not load if file has already been loaded, in which case its value is NIL.

11.2 STORING FILES

(MAKEFILE *FILE OPTIONS REPRINTFNS SOURCEFILE*) [Function]

Makes a new version of the file *FILE*, storing the information specified by *FILE*'s filecoms. Notices *FILE* if not previously noticed (see page 11.12). Then, it adds *FILE* to NOTLISTEDFILES[2] and NOTCOMPILEDFILES.[3]

OPTIONS is a litatom or list of litatoms which specify options. By specifying certain options, MAKEFILE can automatically compile or list *FILE*. Note that if *FILE* does not contain any function definitions, it is not compiled even when *OPTIONS* specifies

[2] Except if *FILE* has on its property list the property FILETYPE with value DON'TLIST, or a list containing DON'TLIST.

[3] Except if *FILE* has on its property list the property FILETYPE with value DON'TCOMPILE, or a list containing DON'TCOMPILE. Also, if *FILE* does not contain any function definitions, it is not added to NOTCOMPILEDFILES, and it is not compiled even when *OPTIONS* specifies C or RC.

C or RC. The options are spelling corrected using the list MAKEFILEOPTIONS. If spelling correction fails, MAKEFILE generates an error. The options are interpreted as follows:

C
RC After making *FILE*, MAKEFILE will compile *FILE* by calling TCOMPL (if C is specified) or RECOMPILE (if RC is specified). If there are any block declarations specified in the filecoms for *FILE*, BCOMPL or BRECOMPILE will be called instead.

If F, ST, STF, or S is the *next* item on *OPTIONS* following C or RC, it is given to the compiler as the answer to the compiler's question LISTING? (see page 12.1). For example, (MAKEFILE 'FOO '(C F LIST)) will dump FOO, then TCOMPL or BCOMPL it specifying that functions are not to be redefined, and finally list the file.

LIST After making *FILE*, MAKEFILE calls LISTFILES to print a hardcopy listing of *FILE*.

CLISPIFY MAKEFILE calls PRETTYDEF with CLISPIFYPRETTYFLG=T (see page 16.20). This causes CLISPIFY to be called on each function defined as an EXPR before it is prettyprinted.[4]

NOCLISP MAKEFILE calls PRETTYDEF with PRETTYTRANFLG=T (see page 16.20). This causes CLISP translations to be printed, if any, in place of the corresponding CLISP expressions, e.g., iterative statements, record expressions, PRINTOUT forms, etc.

FAST MAKEFILE calls PRETTYDEF with PRETTYFLG=NIL (see page 6.54). This causes data objects to be printed rather than prettyprinted, which is much faster.

REMAKE MAKEFILE "remakes" *FILE*: The prettyprinted definitions of functions that have not changed are copied from an earlier version of the symbolic file. Only those functions that have changed are prettyprinted. See page 11.10.

NEW MAKEFILE does *not* remake *FILE*. If MAKEFILEREMAKEFLG=T (the initial setting), the default for all calls to MAKEFILE is to remake. The NEW option can be used to override this default.

REPRINTFNS and *SOURCEFILE* are used when remaking a file, as described on page 11.10.

[4] Alternatively, if *FILE* has the property FILETYPE with value CLISP or a list containing CLISP, PRETTYDEF is called with CLISPIFYPRETTYFLG reset to CHANGES, which will cause CLISPIFY to be called on all functions marked as having been changed. If *FILE* has property FILETYPE with value CLISP, the compiler will DWIMIFY its functions before compiling them (see page 12.9).

If a remake is *not* being performed, MAKEFILE checks the state of *FILE* to make sure that the entire source file was actually LOADed. If *FILE* was loaded as a compiled file, MAKEFILE prints the message CAN'T DUMP: ONLY THE COMPILED FILE HAS BEEN LOADED. Similarly, if only some of the symbolic definitions were loaded via LOADFNS or LOADFROM, MAKEFILE prints CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED. In both cases, MAKEFILE will then ask the user if it should dump anyway; if the user declines, MAKEFILE does not call PRETTYDEF, but simply returns (*FILE* NOT DUMPED) as its value.

The user can indicate that *FILE* must be block compiled together with other files as a unit by putting a list of those files on the property list of each file under the property FILEGROUP. If *FILE* has a FILEGROUP property, the compiler will not be called until all files on this property have been dumped that need to be.

MAKEFILE operates by rebinding PRETTYFLG, PRETTYTRANFLG, and CLISPIFYPRETTYFLG, evaluating each expression on MAKEFILEFORMS (under errorset protection), and then calling PRETTYDEF. The user can add expressions to MAKEFILEFORMS to implement his own options.

(MAKEFILES *OPTIONS* *FILES*) [Function]

Performs (MAKEFILE *FILE* *OPTIONS*) for each file on *FILES* that needs to be dumped. If *FILES*=NIL, FILELST is used. For example, (MAKEFILES 'LIST) will make and list all files that have been changed. In this case, if any typed definitions for any items have been defined or changed and they are *not* contained in one of the files on FILELST, MAKEFILES calls ADDTOFILES? to allow the user to specify where these go. MAKEFILES returns a list of all files that are made.

(CLEANUP $FILE_1$ $FILE_2$ $\cdots$ $FILE_N$) [NLambda NoSpread Function]

Dumps, lists, and recompiles (with RECOMPILE or BRECOMPILE) any of the specified files (unevaluated) requiring the corresponding operation. If no files are specified, FILELST is used. CLEANUP returns NIL.

CLEANUP uses the value of the variable CLEANUPOPTIONS as the *OPTIONS* argument to MAKEFILE. CLEANUPOPTIONS is initially (LIST RC), to indicate that the files should be listed and recompiled. If CLEANUPOPTIONS is set to (RC F), no listing will be performed, and no functions will be redefined as the result of compiling. Alternatively, if $FILE_1$ is a list, it will be interpreted as the list of options regardless of the value of CLEANUPOPTIONS.

(FILES?) [Function]

Prints on the terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of any functions and other typed definitions (if any) that are not contained in any file. If there are any, FILES? then calls ADDTOFILES? to allow the user to specify where these go.

(ADDTOFILES? —) [Function]

Called from MAKEFILES, CLEANUP, and FILES? when there are typed definitions that have been marked as changed which do not belong to any file. ADDTOFILES? lists the names of the changed items, and asks the user if he wants to specify where these items should be put. If user answers N(o), ADDTOFILES? returns NIL without taking any action. If the user answers], this is taken to be an answer to each question that would be asked, and all the changed items are marked as dummy items to be ignored. Otherwise, ADDTOFILES? prints the name of each changed item, and accepts one of the following responses:

A file name or a variable whose value is a list

Adds the item to the corresponding file or list, using ADDTOFILE.

If the item is not the name of a file on FILELST, the user will be asked whether it is a new file. If he says no, then ADDTOFILES? will check whether the item is the name of a list, i.e. whether its value is a list. If not, the user will be asked whether it is a new list.

line-feed

Same as the user's previous response.

space or carriage return

Take no action.

] The item is marked as a dummy item by adding it to NILCOMS. This tells the file package simply to ignore this item.

[The "definition" of the item in question is prettyprinted to the terminal, and then the user is asked again about its disposition.

(ADDTOFILES? prompts with "LISTNAME: (", the user types in the name of a list, i.e. a variable whose value is a list, terminated by a). The item will then only be added to (under) a command in which the named list appears as a filevar. If none are found, a message is printed, and the user is asked again. For example, the user defines a new function FOO3, and when asked where it goes, types (FOOFNS). If the command (FNS * FOOFNS) is found, FOO3 will be added to the value of FOOFNS. If instead the user types (FOOCOMS), and the command (COMS * FOOCOMS) is found, then FOO3 will be added to a command for dumping functions that is contained in FOOCOMS.

Note: If the named list is not also the name of a file, the user can simply type it in without parenthesis as described above.

@ ADDTOFILES? prompts with "Near: (", the user types in the name of an object, and the item is then inserted in a command for dumping objects (of its type) that contains the indicated name. The item is inserted immediately after the indicated name.

(LISTFILES *FILE*$_1$ *FILE*$_2$ ··· *FILE*$_N$) [NLambda NoSpread Function]

Lists each of the specified files (unevaluated). If no files are given, NOTLISTEDFILES is used. Each file listed is removed from NOTLISTEDFILES if the listing is completed. For each file not found, LISTFILES prints the message "*FILENAME* NOT FOUND" and proceeds to the next file. LISTFILES calls the function LISTFILES1 on each file to be listed. The user can advise or redefine LISTFILES1 for more specialized applications.

(Interlisp-10) LISTFILES uses the function TENEX (page 22.6) to tell the operating system to print the file. LISTFILES calls LISTFILES1 which calls TENEX with (CONCAT 'LIST$ *FILENAME* LISTFILESTR), where LISTFILESTR is initially "ᶜʳ". The user can reset LISTFILESTR to specify subcommands for the list command, or advise or redefine LISTFILES1.

(Interlisp-D) LISTFILES1 is initially defined as EMPRESS (page 18.17).

(COMPILEFILES *FILE$_1$ FILE$_2$ ··· FILE$_N$*) [NLambda NoSpread Function]

Executes the RC and C options of MAKEFILE for each of the specified files (unevaluated). If no files are given, NOTCOMPILEDFILES is used. Each file compiled is removed from NOTCOMPILEDFILES. If *FILE$_1$* is a list, it is interpreted as the *OPTIONS* argument to MAKEFILES. This feature can be used to supply an answer to the compiler's LISTING? question, e.g., (COMPILEFILES (STF)) will compile each file on NOTCOMPILEDFILES so that the functions are redefined without the EXPRs definitions being saved.

(WHEREIS *NAME TYPE FILES FN*) [Function]

TYPE is a file package type. WHEREIS sweeps through all the files on the list *FILES* and returns a list of all files containing *NAME* as a *TYPE*. WHEREIS knows about and expands all file package commands and file package macros. *TYPE*=NIL defaults to FNS (to retrieve function definitions). If *FILES* is not a list, the value of FILELST is used.

If *FN* is given, it should be a function (with arguments *NAME*, *FILE*, and *TYPE*) which is applied for every file in *FILES* that contains *NAME* as a *TYPE*. In this case, WHEREIS returns NIL.

If the WHEREIS package (page 23.40) has been loaded, WHEREIS is redefined so that *FILES*=T means to use the whereis package data base, so WHEREIS will find *NAME* even if the file has not been loaded or noticed. *FILES*=NIL always means use FILELST.

11.2.1 Remaking a Symbolic File

Most of the time that a symbolic file is written using MAKEFILE, only a few of the functions that it contains have been changed since the last time the file was written. Rather than prettprinting all of the functions, it is often considerably faster to "remake" the file, copying the prettprinted definitions of unchanged functions from an earlier version of the symbolic file, and only prettyprinting those functions that have been changed.

MAKEFILE will remake the symbolic file if the REMAKE option is specified. If the NEW option is given, the file is not remade, and all of the functions are prettprinted. The default action is specified by the value of MAKEFILEREMAKEFLG: if T (its initial value), MAKEFILE will remake files unless the NEW option is given; if NIL, MAKEFILE will not remake unless the REMAKE option is given.

Note: If the file has never been loaded or dumped, for example if the filecoms were simply set up in memory, then MAKEFILE will never attempt to remake the file, regardless of the setting of MAKEFILEREMAKEFLG, or whether the REMAKE option was specified.

When MAKEFILE is remaking a symbolic file, the user can explicitly indicate the functions which are to be prettyprinted and the file to be used for copying the rest of the function definitions from via the *REPRINTFNS* and *SOURCEFILE* arguments to MAKEFILE. Normally, both of these arguments are defaulted to NIL. In this case, *REPRINTFNS* will be set to those functions that have been changed since the last version of the file was written. For *SOURCEFILE*, MAKEFILE obtains the full name of the most recent version of the file (that it knows about) from the FILEDATES property of the file, and checks to make sure that the file still exists and has the same file date as that stored on the FILEDATES property. If it does, MAKEFILE uses that file as *SOURCEFILE*. This procedure permits the user to LOAD or LOADFROM a file in a different directory, and still be able to remake the file with MAKEFILE. In the case where the most recent version of the file cannot be found, MAKEFILE will attempt to remake using the *original* version of the file (i.e., the one first loaded), specifying as *REPRINTFNS* the union of all changes that have been made since the file was first loaded, which is obtained from the FILECHANGES property of the file. If both of these fail, MAKEFILE prints the message "CAN'T FIND EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF *FILE*, SO IT WILL HAVE TO BE WRITTEN ANEW", and does not remake the file, i.e. will prettyprint all of the functions.

When a remake is specified, MAKEFILE also checks to see how the file was originally loaded (see page 11.12). If the file was originally loaded as a compiled file, MAKEFILE will automatically call LOADVARS to obtain those DECLARE: expressions that are contained on the symbolic file, but not the compiled file, and hence have not been loaded. If the file was loaded by LOADFNS (but not LOADFROM), then LOADVARS will automatically be called to obtain any non-DEFINEQ expressions.

Note: Remaking a symbolic file is considerably faster if the earlier version has a *file map* indicating where the function definitions are located (page 11.38), but it does not depend on this information.

11.3 MARKING CHANGES

The file package needs to know what typed definitions have been changed, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file package operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file package by marking changes. Also, *typed-in* assignment of variables or property values is noticed by the file package. (Note that if a program modifies a variable or property value, this is not noticed.) In some cases the marking procedure can be subtle, e.g. if the user edits a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

The various system functions which create or modify objects call MARKASCHANGED to mark the object as changed. For example, when a function is defined via DEFINE or DEFINEQ, or modified via EDITF, or a DWIM correction, the function is marked as being a changed object of type FNS. Similarly, whenever a new record is declared, or an existing record redeclared or edited, it is marked as being a changed object of type RECORDS, and so on for all of the other file package types.

The user can also call MARKASCHANGED directly to mark objects of a particular file package type as changed:

(MARKASCHANGED *NAME TYPE REASON*) [Function]

Marks *NAME* of type *TYPE* as being changed. *REASON* is a litatom that indicated how *NAME* was changed. MARKASCHANGED recognizes the following values for *REASON*:

DEFINED Used to indicate the creation of *NAME*, e.g. from DEFINE.

CHANGED Used to indicate a change to *NAME*, e.g. from the editor.

DELETED — Used to indicate the deletion of *NAME*, e.g. by DELDEF.

CLISP — Used to indicate the modification of *NAME* by CLISP translation.

For backwards compatibility, MARKASCHANGED also accepts a *REASON* of T (=DEFINED) and NIL (=CHANGED). New programs should avoid using these values.

MARKASCHANGED returns *NAME*. MARKASCHANGED is undoable.

(UNMARKASCHANGED *NAME TYPE*) [Function]

Unmarks *NAME* of type *TYPE* as being changed. Returns *NAME* if *NAME* was marked as changed and is now unmarked, NIL otherwise. UNMARKASCHANGED is undoable.

(FILEPKGCHANGES *TYPE LST*) [NoSpread Function]

If *LST* is not specified (as opposed to being NIL), returns a list of those objects of type *TYPE* that have been marked as changed but not yet associated with their corresponding files (See page 11.14). If *LST* is specified, FILEPKGCHANGES sets the corresponding list. (FILEPKGCHANGES) returns a list of *all* objects marked as changed as a list of elements of the form (*TYPENAME* . *CHANGEDOBJECTS*).

Some properties (e.g. EXPR, ADVICE, MACRO, I.S.OPR, etc..) are used to implement other file package types. For example, if the user changes the value of the property I.S.OPR, he is really changing an object of type I.S.OPR, and the effect is the same as though he had redefined the i.s.opr via a direct call to the function I.S.OPR. If a property whose value has been changed or added does not correspond to a specific file package type, then it is marked as a changed object of type PROPS whose *name* is (*VARIABLENAME PROPNAME*) (except if the property name has a property PROPTYPE with value IGNORE).

Similarly, if the user changes a variable which implements the file package type ALISTS (as indicated by the appearance of the property VARTYPE with value ALIST on the variable's property list), only those entries that are actually changed are marked as being changed objects of type ALISTS, and the "name" of the object will be (*VARIABLENAME KEY*) where *KEY* is CAR of the entry on the alist that is being marked. If the variable corresponds to a specific file package type other than ALISTS, e.g. USERMACROS, LISPXMACROS, etc., then an object of that type is marked. In this case, the name of the changed object will be CAR of the corresponding entry on the alist. For example, if the user edits LISPXMACROS and changes a definition for PL, then the object PL of type LISPXMACROS is marked as being changed.

11.4 NOTICING FILES

Already existing files are "noticed" by LOAD or LOADFROM (or by LOADFNS or LOADVARS when the *VARS* argument is T. New files are noticed when they are constructed by MAKEFILE, or when definitions are first associated with them via FILES? or ADDTOFILES?. Noticing a file updates certain lists and properties so that the file package functions know to include the file in their operations. For example, CLEANUP will only dump files that have been noticed.

The file package uses information stored on the property list of the root name of noticed files. The following property names are used:

FILE [Property Name]

When a file is noticed, the property FILE, value ((*FILECOMS* . *LOADTYPE*)) is added to the property list of its root name. *FILECOMS* is the variable containing the filecoms of the file (see page 11.21). *LOADTYPE* indicates *how* the file was loaded, e.g., completely loaded, only partially loaded as with LOADFNS, loaded as a compiled file, etc.

The property FILE is used to determine whether or not the corresponding file has been modified since the last time it was loaded or dumped. CDR of the FILE property records by type those items that have been changed since the last MAKEFILE. Whenever a file is dumped, these items are moved to the property FILECHANGES, and CDR of the FILE property is reset to NIL.

FILECHANGES [Property Name]

The property FILECHANGES contains a list of all changed items since the file was loaded (there may have been several sequences of editing and rewriting the file). When a file is dumped, the changes in CDR of the FILE property are added to the FILECHANGES property.

FILEDATES [Property Name]

The property FILEDATES contains a list of version numbers and corresponding file dates for this file. These version numbers and dates are used for various integrity checks in connection with *re*making a file (see page 11.10).

FILEMAP [Property Name]

The property FILEMAP is used to store the filemap for the file (see page 11.38). This is used to directly load individual functions from the middle of a file.

To compute the root name, ROOTFILENAME is applied to the name of the file as indicated in the FILECREATED expression appearing at the front of the file, since this name corresponds to the name the file was originally made under. The file package detects that the file being noticed is a compiled file (regardless of its name), by the appearance of more than one FILECREATED expressions. In this case, each of the files mentioned in the following FILECREATED expressions are noticed. For example, if the user performs (BCOMPL '(FOO FIE)), and subsequently loads FOO.DCOM, both FOO and FIE will be noticed.

When a file is noticed, its root name is added to the list FILELST:

FILELST [Variable]

Contains a list of the root names of the files that have been noticed.

LOADEDFILELST [Variable]

Contains a list of the actual names of the files as loaded by LOAD, LOADFNS, etc. For example, if the user performs (LOAD '<NEWLISP>EDITA.COM;3), EDITA will be added to FILELST, but <NEWLISP>EDITA.COM;3 is added to LOADEDFILELST. LOADEDFILELST is not used by the file package; it is maintained solely for the user's benefit.

11.5 DISTRIBUTING CHANGE INFORMATION

Periodically, the function UPDATEFILES is called to find which file(s) contain the elements that have been changed. UPDATEFILES is called by FILES?, CLEANUP, and MAKEFILES, i.e., any procedure that requires the FILE property to be up to date. This procedure is followed rather than update the FILE property after each change because scanning FILELST and examining each file package command can be a time-consuming process, and is not so noticeable when performed in conjunction with a large operation like loading or writing a file.

UPDATEFILES operates by scanning FILELST and interrogating the file package commands for each file. When (if) any files are found that contain the corresponding typed definition, the name of the element is added to the value of the property FILE for the corresponding file. Thus, after UPDATEFILES has completed operating, the files that need to be dumped are simply those files on FILELST for which CDR of their FILE property is non-NIL. For example, if the user loads the file FOO containing definitions for FOO1, FOO2, and FOO3, edits FOO2, and then calls UPDATEFILES, (GETPROP 'FOO 'FILE) will be ((FOOCOMS . T) (FNS FOO2)). If any objects marked as changed have not been transferred to the FILE property for some file, e.g., the user defines a new function but forgets (or declines) to add it to the file package commands for the corresponding file, then both FILES? and CLEANUP will print warning messages, and then call ADDTOFILES? to permit the user to specify on which files these items belong.

The user can also invoke UPDATEFILES directly:

(UPDATEFILES — —) [Function]

(UPDATEFILES) will update the FILE properties of the noticed files.

11.6 FILE PACKAGE TYPES

In addition to the definitions of functions and values of variables, source files in Interlisp can contain a variety of other information, e.g. property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file package uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a litatom), a definition of a given type (called the file package type). Note that the same name may have several definitions of different types. For example, a litatom may have both a function definition and a variable definition. The file package also keeps track of the file that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

A file package type is an abstract notion of a class of objects which share the property that every object of the same file package type is stored, retrieved, edited, copied etc., by the file package in the same way. Each file package type is identified by a litatom, which can be given as an argument to the functions that manipulate typed definitions. The user may define new file package types, as described in page 11.20.

FILEPKGTYPES [Variable]

The value of FILEPKGTYPES is a list of all file package types, including any that may have been defined by the user.

The file package is initialized with the following built-in file package types:

FNS Function definitions.

VARS (top-level) Variable values.

PROPS Property name/value pairs. When a property is changed or added, an object of type PROPS, with "name" (*LITATOM PROPNAME*) is marked as being changed.

Note that some properties are used to implement other file package types. For example, the property MACRO implements the file package type MACROS, the property ADVICE implements ADVICE, etc. This is indicated by putting the property PROPTYPE, with value of the file package type on the property list of the property name. For example, (GETPROP 'MACRO 'PROPTYPE) => MACROS. When such a property is changed or added, an object of the corresponding file package type is marked. If (GETPROP *PROPNAME* 'PROPTYPE) => IGNORE, the change is ignored. The FILE, FILEMAP, FILEDATES, etc. properties are all handled this way. (Note that IGNORE cannot be the name of a file package type implemented as a property).

ALISTS Alists (association lists); a list of dotted pairs accessed via ASSOC and PUTASSOC.

A variable is declared to have an association list as its value by putting on its property list the property VARTYPE with value ALIST. In this case, each dotted pair on the list is an object of type ALISTS. When the value of such a variable is changed, only those entries in the a-list that are actually changed or added are marked as changed objects of type ALISTS (with "name" (*LITATOM KEY*)). Objects of type ALISTS are dumped via the ALISTS or ADDVARS file package commands.

Note that some alists are used to "implement" other file package types. For example, the value of the global variable USERMACROS implements the file package type USERMACROS and the values of LISPXMACROS and LISPXHISTORYMACROS implement the file package type LISPXMACROS. This is indicated by putting on the property list of the variable the property VARTYPE with value a list of the form (ALIST *FILEPKGTYPE*). For example, (GETPROP 'LISPXHISTORYMACROS 'VARTYPE) => (ALIST LISPXMACROS).

EXPRESSIONS Expressions.

Objects of type EXPRESSIONS are written out via the P file package command, and marked as being changed via the REMEMBER programmers assistant command (page 8.13).

MACROS Compiler macros. See page 5.17.

USERMACROS User edit macros. See page 17.48.

LISPXMACROS (values in) LISPXMACROS and LISPXHISTORYMACROS. See page 8.19.

ADVICE Advice. See page 10.7.

FILEPKGCOMS File package commands/types. New file package types and commands can be defined as explained on page 11.20 and page 11.32.

RECORDS Record declarations. See page 3.1..

FIELDS Fields of records. The "definition" of an object of type FIELDS is a list of all the record declarations which contain the name. See page 3.1.

I.S.OPRS Iterative statement operators. See page 4.5.

TEMPLATES Masterscope templates. See page 13.1.

FILES Files. Files may be treated like other typed definitions.

FILEVARS Filevars. See page 11.30.

11.6.1 Functions for Manipulating Typed Definitions

The functions described below can be used to manipulate typed definitions, without needing to know how the manipulations are done. For example, (GETDEF 'FOO 'FNS) will return the function definition of FOO, (GETDEF 'FOO 'VARS) will return the variable value of FOO, etc. All of the functions use the following conventions:

(1) Any argument that expects a list of litatoms will also accept a single litatom, operating as though it were enclosed in a list. For example, if the argument *FILES* should be a list of files, it may also be a single file.

(2) *TYPE* is a file package type. *TYPE*=NIL is equivalent to *TYPE*=FNS. The singular form of a file package type is also recognized, e.g. *TYPE*=VAR is equivalent to *TYPE*=VARS.

(3) *FILES*=NIL is equivalent to *FILES*=FILELST.

(4) *SOURCE* is used to indicate the source of a definition, that is, where the definition should be found. *SOURCE* can be one of:

CURRENT Get the definition currently in effect.

SAVED Get the "saved" definition, as stored by SAVEDEF (page 11.18).

FILE . Get the definition contained on the (first) file determined by WHEREIS (page 11.10).

Note: WHEREIS is called with *FILES*=T, so that if the WHEREIS package (page 23.40) is loaded, the WHEREIS data base will be used to find the file containing the definition.

? Get the definition currently in effect if there is one, else the saved definition if there is one, otherwise the definition from a file determined by WHEREIS. Like specifying CURRENT, SAVED, and FILE in order, and taking the first definition that is found.

a file name or list of file names
Get the definition from the first of the indicated files that contains one.

NIL In most cases, giving *SOURCE*=NIL (or not specifying it at all) is the same as giving ?, to get either the current, saved, or filed definition. However, with HASDEF, *SOURCE*=NIL is interpreted as equal to *SOURCE*=CURRENT, which only tests if there is a current definition.

(5) All functions which make destructive changes are undoable.

The operation of most of the functions described below can be changed or extended by modifying the appropriate properties for the corresponding file package type using the function FILEPKGTYPE, described on page 11.20.

(GETDEF *NAME TYPE SOURCE OPTIONS*) [Function]

Returns the definition of *NAME*, of type *TYPE*, from *SOURCE*. For most types, GETDEF returns the expression which would be prettyprinted when dumping *NAME* as *TYPE*. For example, for *TYPE*=FNS, an EXPR definition is returned, for *TYPE*=VARS, the value of *NAME* is returned, etc.

OPTIONS is a list which specifies certain options:

NOERROR — GETDEF causes an error if an appropriate definition cannot be found, unless *OPTIONS* is or contains NOERROR.

a string — If *OPTIONS* is or contains a string, that string will be returned if no definition is found. The caller can thus determine whether a definition was found, even for types for which NIL or NOBIND are acceptable definitions.

NOCOPY — GETDEF returns a copy of the definition unless *OPTIONS* is or contains NOCOPY.

NODWIM — A FNS definition will be dwimified if it is likely to contain CLISP unless *OPTIONS* is or contains NODWIM.

(PUTDEF *NAME TYPE DEFINITION*) [Function]

Defines *NAME* of type *TYPE* with *DEFINITION*. For *TYPE*=FNS, does a DEFINE; for *TYPE*=VARS, does a SAVESET, etc.

For *TYPE*=FILES, PUTDEF establishes the command list, notices *NAME*, and then calls MAKEFILE to actually dump the file *NAME*, copying functions if necessary from the "old" file (supplied as part of *DEFINITION*).

(HASDEF *NAME TYPE SOURCE SPELLFLG*) [Function]

Returns *NAME* if *NAME* is the name of something of type *TYPE*. If not, attempts spelling correction if *SPELLFLG*=T, and returns the spelling-corrected *NAME*. Otherwise returns NIL.

(HASDEF NIL *TYPE*) returns T if NIL has a valid definition.

Note: if *SOURCE*=NIL, *HASDEF* interprets this as equal to *SOURCE*=CURRENT, which only tests if there is a current definition.

(TYPESOF *NAME POSSIBLETYPES IMPOSSIBLETYPES SOURCE*) [Function]

Returns a list of the types in *POSSIBLETYPES* but not in *IMPOSSIBLETYPES* for which *NAME* has a definition. FILEPKGTYPES is used if *POSSIBLETYPES* is NIL.

(COPYDEF *OLD NEW TYPE SOURCE OPTIONS*) [Function]

Defines *NEW* to have a copy of the definition of *OLD* by doing PUTDEF on a copy of the definition retrieved by (GETDEF *OLD TYPE SOURCE OPTIONS*). *NEW* is substituted for *OLD* in the copied definition, in a manner that may depend on the *TYPE*.

For example, (COPYDEF 'PDQ 'RST 'FILES) sets up RSTCOMS to be a copy of PDQCOMS, changes things like (VARS * PDQVARS) to be (VARS * RSTVARS) in RSTCOMS, and performs a MAKEFILE on RST such that the appropriate definitions get copied from PDQ.

Note: COPYDEF disables the NOCOPY option of GETDEF, so *NEW* will always have a *copy* of the definition of *OLD*.

(DELDEF *NAME TYPE*) [Function]

Removes the definition of *NAME* as a *TYPE* that is currently in effect.

(SHOWDEF *NAME TYPE FILE*) [Function]

Prettyprints the definition of *NAME* as a *TYPE* to *FILE*. This shows the user how *NAME* would be written to a file. Used by ADDTOFILES? (page 11.8).

(EDITDEF *NAME TYPE SOURCE EDITCOMS*) [Function]

Edits the definition of *NAME* as a *TYPE*. Essentially performs (PUTDEF *NAME TYPE* (EDITE (GETDEF *NAME TYPE SOURCE*) *EDITCOMS*)).

(SAVEDEF *NAME TYPE DEFINITION*) [Function]

Makes *DEFINITION* (or if *DEFINITION*=NIL, the definition of *NAME* as a *TYPE* that is currently in effect) be the "saved" definition for *NAME* as a *TYPE*. If *TYPE*=FNS (or *TYPE*=NIL), this consists of storing *DEFINITION* on *NAME*'s property list under property EXPR, CODE, or SUBR. For *TYPE*=VARS, the definition is stored as the value of the VALUE property. For other types, *DEFINITION* is stored in an internal data structure, from where it can be retrieved by GETDEF or UNSAVEDEF.

(UNSAVEDEF *NAME TYPE* —) [Function]

Makes the "saved" definition of *NAME* as a *TYPE* be the definition currently in effect. If *TYPE*=FNS (or *TYPE*=NIL), UNSAVEDEF will unsave the EXPR property if any, else CODE or SUBR. UNSAVEDEF also recognizes *TYPE*=EXPR, CODE, or SUBR, meaning to unsave the corresponding definition only.

(LOADDEF *NAME TYPE SOURCE*) [Function]

Equivalent to (PUTDEF *NAME TYPE* (GETDEF *NAME TYPE SOURCE*)). LOADDEF is essentially a generalization of LOADFNS, e.g. it enables loading a single record declaration from a file. Note that (LOADDEF *FN*) will give *FN* an EXPR definition, either obtained from its property list or a file, unless it already has one.

(CHANGECALLERS *OLD NEW TYPES FILES METHOD*) [Function]

Finds all of the places where *OLD* is used as any of the types in *TYPES* and changes those places to use *NEW*. For example, (CHANGECALLERS 'NLSETQ 'ERSETQ) will change all calls to NLSETQ to be calls to ERSETQ. Also changes occurrences of *OLD* to *NEW* inside the filecoms of any file, inside record declarations, properties, etc.

CHANGECALLERS attempts to determine if *OLD* might be used as more than one type; for example, if it is both a function and a record field. If so, rather than performing the transformation *OLD* -> *NEW* automatically, the user is allowed to edit all of the places where *OLD* occurs. For each occurrence of *OLD*, the user is asked whether he wants to make the replacement. If he responds with anything except Yes or No, the editor is invoked on the expression containing that occurrence.

Currently there are two different methods for determining which functions are to be examined. If *METHOD*=EDITCALLERS, EDITCALLERS is used to search *FILES* (see page 17.59). If *METHOD*=MASTERSCOPE, then the Masterscope database is used instead. *METHOD*=NIL defaults to MASTERSCOPE if the value of the variable DEFAULTRENAMEMETHOD is MASTERSCOPE and a Masterscope database exists, otherwise it defaults to EDITCALLERS.

(RENAME *OLD NEW TYPES FILES METHOD*) [Function]

First performs (COPYDEF *OLD NEW TYPE*) for all *TYPE* inside *TYPES*. It then calls CHANGECALLERS to change all occurrences of *OLD* to *NEW*, and then "deletes" *OLD* with DELDEF. For example, if the user has a function FOO which he now wishes to call FIE, he simply performs (RENAME 'FOO 'FIE), and FIE will be given FOO's definition, and all places that FOO are called will be changed to call FIE instead.

(COMPARE *NAME1 NAME2 TYPE SOURCE1 SOURCE2*) [Function]

Compares definiton of *NAME1* with that of *NAME2*, i.e. performs (COMPARELISTS (GETDEF *NAME1 TYPE SOURCE1*) (GETDEF *NAME2 TYPE SOURCE2*))

(COMPAREDEFS *NAME TYPE SOURCES*) [Function]

Calls COMPARELISTS on all pairs of definitions of *NAME* as a *TYPE* obtained from the various *SOURCES*.

11.6.2 Defining New File Package Types

All manipulation of typed definitions in the file package is done using the type-independent functions GETDEF, PUTDEF, etc. Therefore, to define a new file package type, it is only necessary to specify what these functions should do when dealing with a typed definition of the new type. Each file package type has the following properties, whose values are functions or lists of functions:

Note: These functions are defined to take a *TYPE* argument so that the user may have the same function for more than one type.

GETDEF

Value is a function of three arguments, *NAME*, *TYPE*, and *OPTIONS*, which should return the current definition of *NAME* as a type *TYPE*. Used by GETDEF (which passes its *OPTION* argument).

If there is no GETDEF property, a file package command for dumping *NAME* is created (by MAKENEWCOM). This command is then used to write the definition of *NAME* as a type *TYPE* onto the file FILEPKG.SCRATCH (in Interlisp-D, this file is created on the {CORE} device). This expression is then read back in and returned as the current definition.

FILEGETDEF — This enables the user to provide a way of obtaining definitions from a file that is more efficient than the default procedure used by GETDEF. Value is a function of four arguments, *NAME*, *TYPE*, *FILE*, and *OPTIONS*. The function is applied by GETDEF when it is determined that a typed definition is needed from a particular file. The function must open and search the given file and return any *TYPE* definition for *NAME* that it finds.

PUTDEF — Value is a function of three arguments, *NAME*, *TYPE*, and *DEFINITION*, which should store *DEFINITION* as the definition of *NAME* as a type *TYPE*. Used by PUTDEF.

DELDEF — Value is a function of two arguments, *NAME*, and *TYPE*, which removes the definition of of *NAME* as a *TYPE* that is currently in effect. Used by DELDEF.

NEWCOM — Value is a function of four arguments, *NAME*, *TYPE*, *LISTNAME*, and *FILE*. Specifies how to make a new (instance of a) file package command to dump *NAME*, an object of type *TYPE*. The function should return the new file package command. Used by ADDTOFILE and SHOWDEF.

If *LISTNAME* is non-NIL, this means that the user specified *LISTNAME* as the filevar in his interaction with ADDTOFILES? (see page 11.30).

If no NEWCOM is specified, the default is to call DEFAULTMAKENEWCOM, which will construct and return a command of the form (*TYPE NAME*). DEFAULTMAKENEWCOM can be advised or redefined by the user.

WHENCHANGED — Value is a list of functions to be applied to *NAME*, *TYPE*, and *REASON* when *NAME*, an instance of type *TYPE*, is changed or defined (see MARKASCHANGED, page 11.11). Used for various applications, e.g. when an object of type I.S.OPRS changes, it is necessary to clear the corresponding translatons from CLISPARRAY.

The WHENCHANGED functions are called before the object is marked as changed, so that it can, in fact, decide that the object is *not* to be marked as changed, and execute (RETFROM 'MARKASCHANGED).

Note: For backwards compatibility, the *REASON* argument passed to WHENCHANGED functions is either T (for DEFINED) and NIL (for CHANGED).

WHENFILED — Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is added to *FILE*.

WHENUNFILED — Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is removed from *FILE*.

DESCRIPTION — Value is a string which describes instances of this type. For example, for type RECORDS, the value of DESCRIPTION is the string "record declarations".

The function FILEPKGTYPE is used to define new file package types, or to change the attributes of existing types. Note that it is possible to redefine the attributes of system file package types, such as FNS or PROPS.

(FILEPKGTYPE *TYPE* *PROP*$_1$ *VAL*$_1$ ··· *PROP*$_N$ *VAL*$_N$) [NoSpread Function]

Nospread function for defining new file package types, or changing attributes of existing file package types. *PROP*$_i$ is one of the property names given above; *VAL*$_i$

is the value to be given to that property. Returns *TYPE*.

(FILEPKGTYPE *TYPE PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGTYPE *TYPE* returns an alist of all of the defined properties of *TYPE*, using the property names as keys.

11.7 FILE PACKAGE COMMANDS

The basic mechanism for creating symbolic files is the function MAKEFILE (page 11.6). For each file, the file package has a data structure known as the "filecoms", which specifies what typed descriptions are contained in the file. A filecoms is a list of file package commands, each of which specifies objects of a certain file package type which should be dumped. For example, the filecoms

```
(  (FNS FOO)
   (VARS FOO BAR BAZ)
   (RECORDS XYZZY)  )
```

has a FNS, a VARS, and a RECORDS file package command. This filecoms specifies that the function definition for FOO, the variable values of FOO, BAR, and BAZ, and the record declaration for XYZZY should be dumped.

By convention, the filecoms of a file *x* is stored as the value of the litatom *x*COMS. For example, (MAKEFILE 'FOO.;27) will use the value of FOOCOMS as the filecoms. This variable can be directly manipulated, but the file package contains facilities which make constructing and updating filecoms easier, and in some cases automatic (See page 11.32).

A file package command is an instruction to MAKEFILE to perform an explicit, well-defined operation, usually printing an expression. Usually there is a one-to-one correspondence between file package types and file package commands; for each file package type, there is a file package command which is used for writing objects of that type to a file, and each file package command is used to write objects of a particular type. However, in some cases, the same file package type can be dumped by several different file package commands. For example, the file package commands PROP, IFPROP, and PROPS all dump out objects with the file package type PROPS. This means if the user changes an object of file package type PROPS via EDITP, a typed-in call to PUTPROP, or via an explicit call to MARKASCHANGED, this object can be written out with any of the above three commands. Thus, when the file package attempts to determine whether this typed object is contained on a particular file, it must look at instances of all three file package commands PROP, IFPROP, and PROPS, to see if the corresponding atom and property are specified. It is also permissible for a single file package command to dump several different file package types. For example, the user can define a file package command which dumps both a function definition and its macro. Conversely, some file package comands do not dump any file package types at all, such as the E command.

For each file package command, the file package must be able to determine what typed definitions the command will cause to be printed so that the file package can determine on what file (if any) an object of a given type is contained (by searching through the filecoms). Similarly, for each file package type, the file package must be able to construct a command that will print out an object of that type. In other words, the file package must be able to map file package commands into file package types, and vice versa. Information can be provided to the file package about a particular file package command via the function FILEPKGCOM (page 11.32), and information about a particular file package type via the function FILEPKGTYPE (page 11.20). In the absence of other information, the default is simply that a file package command of the form (*X* *NAME*) prints out the definition of *NAME* as a type *X*, and, conversely, if *NAME* is an object of type *X*, then *NAME* can be written out by a command of the form (*X* *NAME*).

If a file package function is given a command or type that is not defined, it attempts spelling correction[5] using FILEPKGCOMSPLST as a spelling list. If successful, the corrected version of the list of file package commands is written (again) on the output file.[6] If unsuccessful, generates an error, BAD FILE PACKAGE COMMAND.

File package commands can be used to save on the output file definitions of functions, values of variables, property lists of atoms, advised functions, edit macros, record declarations, etc. The interpretation of each file package command is as follows:

(FNS $FN_1 \cdots FN_N$) [File Package Command]

Writes a DEFINEQ expression with the function definitions of $FN_1 \cdots FN_N$.

The user should never print a DEFINEQ expression directly onto a file himself (by using the P file package command, for example), because MAKEFILE generates the filemap of function definitions from the FNS file package commands (see page 11.38).

(VARS $VAR_1 \cdots VAR_N$) [File Package Command]

For each VAR_i, writes an expression to set its top level value when the file is loaded. If VAR_i is atomic, VARS writes out an expression to set VAR_i to the top-level value it had at the time the file was written. If VAR_i is non-atomic, it is interpreted as (*VAR* *FORM*), and VARS write out an expression to set *VAR* to the value of *FORM* (evaluated when the file is loaded).

VARS prints out expressions using RPAQQ and RPAQ, which are like SETQQ and SETQ except that they also perform some special operations with respect to the file package (see page 11.37).

Note: VARS cannot be used for putting arbitrary variable values on files. For example, if the value of a variable is an array (or many other data types), a litatom which represents the array is dumped in the file instead of the array itself. The HORRIBLEVARS file package command (page 11.25) provides a way of saving and reloading variables whose values contain re-entrant or circular list structure, user data types, arrays, or hash arrays.

(INITVARS $VAR_1 \cdots VAR_N$) [File Package Command]

INITVARS is used for initializing variables, setting their values only when they are currently NOBIND. A variable value defined in an INITVARS command will not change an already established value. This means that re-loading files to get some other information will not automatically revert to the initialization values.

[5] unless DWIMFLG or NOSPELLFLG=NIL. See page 15.12.

[6] since at this point, the uncorrected list of file package commands would already have been printed on the output file. When the file is loaded, this will result in *FILE*COMS being reset, and may cause a message to be printed, e.g., (FOOCOMS RESET). The value of FOOCOMS would then be the corrected version.

The format of an INITVARS command is just like VARS. The only difference is that if *VAR*$_i$ is atomic, the current value is not dumped; instead NIL is defined as the initialization value. Therefore, (INITVARS FOO (FUM 2)) is the same as (VARS (FOO NIL)(FUM 2)), if FOO and FUM are both NOBIND.

INITVARS writes out an RPAQ? expression on the file instead of RPAQ or RPAQQ.

(ADDVARS (*VAR*$_1$. *LST*$_1$) ··· (*VAR*$_N$. *LST*$_N$)) [File Package Command]

For each (*VAR*$_i$. *LST*$_i$), writes an ADDTOVAR to add each element of *LST*$_i$ to the list that is the value of *VAR*$_i$ at the time the file is loaded. The new value of *VAR*$_i$ will be the union of its old value and *LST*$_i$. If the value of *VAR*$_i$ is NOBIND, it is first set to NIL.

For example, (ADDVARS (DIRECTORIES LISP LISPUSERS)) will add LISP and LISPUSERS to the value of DIRECTORIES.

If *LST*$_i$ is not specified, *VAR*$_i$ is initialized to NIL if its current value is NOBIND. In other words, (ADDVARS (*VAR*)) will initialize *VAR* to NIL if *VAR* has not previously been set.

(ALISTS (*VAR*$_1$ *KEY*$_1$ *KEY*$_2$ ···) ··· (*VAR*$_N$ *KEY*$_3$ *KEY*$_4$ ···)) [File Package Command]

VAR$_i$ is a variable whose value is an alist, such as EDITMACROS, BAKTRACELST, etc. For each *VAR*$_i$, ALISTS writes out expressions which will restore the values associated with the specified keys. For example, (ALISTS (BREAKMACROS BT BTV)) will dump the definition for the BT and BTV commands on BREAKMACROS.

Some alists (USERMACROS, LISPXMACROS, etc.) are used to implement other file package types, and they have their own file package commands.

(PROP *PROPNAME* *LITATOM*$_1$ ··· *LITATOM*$_N$) [File Package Command]

Writes a PUTPROPS expression to restore the value of the *PROPNAME* property of each litatom *LITATOM*$_i$ when the file is loaded.

If *PROPNAME* is a list, expressions will be written for each property on that list. If *PROPNAME* is the litatom ALL, the values of all user properties (on the property list of each *LITATOM*$_i$) are saved. SYSPROPS is a list of properties used by system functions. Only properties *not* on that list are dumped when the ALL option is used.

If *LITATOM*$_i$ does not have the property *PROPNAME* (as opposed to having the property with value NIL), a warning message "NO *PROPNAME* PROPERTY FOR *LITATOM*$_i$" is printed. The command IFPROP can be used if it is not known whether or not an atom will have the corresponding property.

(IFPROP *PROPNAME* *LITATOM*$_1$ ··· *LITATOM*$_N$) [File Package Command]

Same as the PROP file package command, except that it only saves the properties that actually appear on the property list of the corresponding atom. For example, if FOO1 has property PROP1 and PROP2, FOO2 has PROP3, and FOO3 has property PROP1 and PROP3, then (IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3) will save only those five property values.

(PROPS (*LITATOM*$_1$ *PROPNAME*$_1$) ··· (*LITATOM*$_N$ *PROPNAME*$_N$)) [File Package Command]

Similar to PROP command. Writes a PUTPROPS expression to restore the value of *PROPNAME*$_i$ for each *LITATOM*$_i$ when the file is loaded.

As with the PROP command, if *LITATOM*$_i$ does not have the property *PROPNAME* (as opposed to having the property with NIL value), a warning message "NO *PROPNAME*$_i$ PROPERTY FOR *LITATOM*$_i$" is printed.

(P *EXP*$_1$ ··· *EXP*$_N$) [File Package Command]

Writes each of the expressions *EXP*$_1$ ··· *EXP*$_N$ on the output file, where they will be evaluated when the file is loaded.

(E *FORM*$_1$ ··· *FORM*$_N$) [File Package Command]

Each of the forms *FORM*$_1$ ··· *FORM*$_N$ is evaluated at *output* time, when MAKEFILE interpretes this file package command.

(COMS *COM*$_1$ ··· *COM*$_N$) [File Package Command]

Each of the commands *COM*$_1$ ··· *COM*$_N$ is interpreted as a file package command.

(* . *TEXT*) [File Package Command]

Used for inserting comment in a file. The file package command is simply written on the output file; it will be ignored when the file is loaded.

If the first element of *TEXT* is another *, a form-feed is printed on the file before the comment.

(ADVISE *FN*$_1$ ··· *FN*$_N$) [File Package Command]

For each function *FN*$_i$, writes expressions to reinstate the function to its advised state when the file is loaded. See page 10.7.

(ADVICE *FN*$_1$ ··· *FN*$_N$) [File Package Command]

For each function *FN*$_i$, writes a PUTPROPS expression which will put the advice back on the property list of the function. The user can then use READVISE to reactivate the advice.

(USERMACROS *LITATOM*$_1$ ··· *LITATOM*$_N$) [File Package Command]

Each litatom *LITATOM*$_i$ is the name of a user edit macro. Writes expressions to add the edit macro definitions of *LITATOM*$_i$ to USERMACROS, and adds the names of the commands to the appropriate spelling lists.

If *LITATOM*$_i$ is not a user macro, a warning message "no EDIT MACRO for *LITATOM*$_i$" is printed.

(FILEPKGCOMS *LITATOM*$_1$ ··· *LITATOM*$_N$) [File Package Command]

Each litatom *LITATOM*$_i$ is either the name of a user-defined file package command or a user-defined file package type (or both). Writes expressions which will restore each command/type.

If *LITATOM*$_i$ is not a file package command or type, a warning message "no FILE PACKAGE COMMAND for *LITATOM*$_i$" is printed.

(LISPXMACROS *LITATOM*$_1$ ··· *LITATOM*$_N$) [File Package Command]

Each *LITATOM*$_i$ is defined on LISPXMACROS or LISPXHISTORYMACROS (see page

8.19). Writes expressions which will save and restore the definition for each macro, as well as making the necessary additions to LISPXCOMS

(RECORDS *REC_1* ··· *REC_N*) [File Package Command]

Each *REC_i* is the name of a record (see page 3.1). Writes expressions which will redeclare the records when the file is loaded.

(INITRECORDS *REC_1* ··· *REC_N*) [File Package Command]

Similar to RECORDS, INITRECORDS writes expressions on a file that will, when loaded, perform whatever initialization/allocation is necessary for the indicated records. However, the record declarations themselves are not written out. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the allocation for these records must still be done.

(I.S.OPRS *OPR_1* ··· *OPR_N*) [File Package Command]

Each *OPR_i* is the name of a user-defined i.s.opr (see page 4.13). Writes expressions which will redefine the i.s.oprs when the file is loaded.

(TEMPLATES *$LITATOM_1$* ··· *$LITATOM_N$*) [File Package Command]

Each *$LITATOM_i$* is a litatom which has a Masterscope template (see page 13.18). Writes expressions which will restore the templates when the file is loaded.

(BLOCKS *$BLOCK_1$* ··· *$BLOCK_N$*) [File Package Command]

For each *$BLOCK_i$*, writes a DECLARE: expression which the block compile functions interpret as a block declaration. See page 12.14.

(MACROS *$LITATOM_1$* ··· *$LITATOM_N$*) [File Package Command]

Each *$LITATOM_i$* is a litatom with a MACRO definition (and/or a DMACRO, 10MACRO, etc.). Writes out an expression to restore all of the macro properties for each *$LITATOM_i$*, embedded in a DECLARE: EVAL@COMPILE so the macros will be defined when the file is compiled. See page 5.17.

(SPECVARS *VAR_1* ··· *VAR_N*) [File Package Command]
(LOCALVARS *VAR_1* ··· *VAR_N*) [File Package Command]
(GLOBALVARS *VAR_1* ··· *VAR_N*) [File Package Command]

Outputs the corresponding compiler declaration embedded in a DECLARE: DOEVAL@COMPILE DONTCOPY. See page 12.3.

(UGLYVARS *VAR_1* ··· *VAR_N*) [File Package Command]

Like VARS, except that the value of each *VAR_i* may contain structures for which READ is not an inverse of PRINT, e.g. arrays, readtables, user data types, etc. Uses HPRINT (page 6.24).

(HORRIBLEVARS *VAR_1* ··· *VAR_N*) [File Package Command]

Like UGLYVARS, except structures may also contain circular pointers. Uses HPRINT (page 6.24). The values of *VAR_1* ··· *VAR_N* are printed in the same operation, so that they may contain pointers to common substructures.

UGLYVARS does not do any checking for circularities, which results in a large speed and internal-storage advantage over HORRIBLEVARS. Thus, if it is known that the data structures do *not* contain circular pointers, UGLYVARS should be used instead of HORRIBLEVARS.

(DECLARE: . *FILEPKGCOMS/FLAGS*) [File Package Command]

Normally expressions written onto a symbolic file are (1) evaluated when loaded; (2) copied to the compiled file when the symbolic file is compiled (see page 12.1); and (3) not evaluated at compile time. DECLARE: allows the user to override these defaults.

FILEPKGCOMS/FLAGS is a list of file package commands, possibly interspersed with "tags". The output of those file package commands within *FILEPKGCOMS/FLAGS* is embedded in a DECLARE: expression, along with any tags that are specified. For example, (DECLARE: EVAL@COMPILE DONTCOPY (FNS ···) (PROP ···)) would produce (DECLARE: EVAL@COMPILE DONTCOPY (DEFINEQ ···) (PUTPROPS ···)). DECLARE: is *defined* as an nlambda nospread function, which processes its arguments by evaluating or not evaluating each expression depending on the setting of internal state variables. The initial setting is to evaluate, but this can be overridden by specifying the DONTEVAL@LOAD tag.

DECLARE: expressions are specially processed by the compiler. For the purposes of compilation, DECLARE: has two principal applications: (1) to specify forms that are to be evaluated at compile time, presumably to affect the compilation, e.g., to set up macros; and/or (2) to indicate which expressions appearing in the symbolic file are *not* to be copied to the output file. (Normally, expressions are *not* evaluated and *are* copied.) Each expression in CDR of a DECLARE: form is either evaluated/not-evaluated and copied/not-copied depending on the settings of two internal state variables, initially set for copy and not-evaluate. These state variables can be reset for the remainder of the expressions in the DECLARE: by means of the tags DONTCOPY, EVAL@COMPILE, etc.

The tags are:

EVAL@LOAD
DOEVAL@LOAD — Evaluate the following forms when the file is loaded (unless overridden by DONTEVAL@LOAD).

DONTEVAL@LOAD — Do not evaluate the following forms when the file is loaded.

EVAL@LOADWHEN — This tag can be used to provide conditional evaluation. The value of the expression immediately following the tag determines whether or not to evaluate subsequent expressions when loading. ··· EVAL@LOADWHEN T ··· is equivalent to ··· EVAL@LOAD ···

COPY
DOCOPY — When compiling, copy the following forms into the compiled file.

DONTCOPY — When compiling, do not copy the following forms into the compiled file.

COPYWHEN — When compiling, if the next form evaluates to non-NIL, copy the following forms into the compiled file.

EVAL@COMPILE
DOEVAL@COMPILE When compiling, evaluate the following forms.

DONTEVAL@COMPILE
When compiling, do not evaluate the following forms.

EVAL@COMPILEWHEN
When compiling, if the next form evaluates to non-NIL, evaluate the following forms.

FIRST For expressions that are to be copied to the compiled file, the tag FIRST can be used to specify that the following expressions in the DECLARE: are to appear at the front of the compiled file, before anything else except the FILECREATED expressions (see page 11.35). For example, (DECLARE: COPY FIRST (P (PRINT *MESS1* T)) NOTFIRST (P (PRINT *MESS2* T))) will cause (PRINT *MESS1* T) to appear first in the compiled file, followed by any functions, then (PRINT *MESS2* T).

NOTFIRST Reverses the effect of FIRST.

The value of DECLARETAGSLST is a list of all the tags used in DECLARE: expressions. If a tag not on this list appears in a DECLARE: file package command, performs spelling correction using DECLARETAGSLST as a spelling list.

Note that the function LOADCOMP (page 11.6) provides a convenient way of obtaining information from the DECLARE: expressions in a file, without reading in the entire file. This information may be used for compiling other files.

(EXPORT $COM_1 \cdots COM_N$) [File Package Command]
This command is used for "exporting" definitions. Like COM, each of the commands $COM_1 \cdots COM_N$ is interpreted as a file package command. The commands are also flagged in the file as being "exported" commands, for use with GATHEREXPORTS (see page 11.29).

(CONSTANTS $VAR_1 \cdots VAR_N$) [File Package Command]
Like VARS, for each VAR_i writes an expression to set its top level value when the file is loaded. Also writes a CONSTANTS expression to declare these variables as constants (see page 12.6). Both of these expressions are wrapped in a (DECLARE: EVAL@COMPILE ···) expression, so they can be used by the compiler.

Like VARS, VAR_i can be non-atomic, in which case it is interpreted as (*VAR FORM*), and passed to CONSTANTS (along with the variable being initialized to *FORM*).

(ORIGINAL $COM_1 \cdots COM_N$) [File Package Command]
Each of the commands COM_i will be interpreted as a file package command without regard to any file package macros (as defined by the MACRO property of the FILEPKGCOM function, page 11.32). Useful for redefining a built-in file package command in terms of itself.

Note that some of the "built-in" file package commands are defined by file package macros, so interpreting them (or new user-defined file package commands) with ORIGINAL will fail. ORIGINAL was never intended to be used outside of a file package command macro.

(FILES . *FILES/LISTS*) [File Package Command]

Used to specify auxiliary files to be loaded in when the file is loaded. *FILES/LISTS* is a list of files, possibly interspersed with lists, which may be used to specify certain loading options. Within these lists, the following tokens are recognized:

The elements of the FILES command are the (namefield) of the files to load. There are actually several other ways to load in files; the FILES command interprets LISTP elements of the commands as a series of tokens which change its state. Those tokens can be:

FROM *DIRECTORY* Pack the given directory onto the beginning of the file. For example, (FILES (FROM LISPUSERS) CJSYS). If this is not specified, the default is to use the same directory as the file containing the FILES command.

SOURCE Load the source version of the file rather than the compiled version.

COMPILED Load the compiled version of the file (the default).

LOAD Load the file with by calling LOAD? (the default).

LOADCOMP Load the file with LOADCOMP? rather than LOAD?. Automatically implies SOURCE.

LOADFROM Load the file with LOADFROM rather than LOAD?.

SYSLOAD Load the file with *LDFLG*=SYSOUT. This is mainly used when loading system files.

PROP Load the file with *LDFLG*=PROP, so function definitions loaded will be stored on property lists.

ALLPROP Load the file with *LDFLG*=ALLPROP, so both function definitions and variable values loaded will be stored on property lists.

These tokens can be joined together in a single list. For example, an actual command in the FTP package is:

```
(FILES (LOADCOMP) NET (SYSLOAD FROM LISPUSERS) CJSYS)
```

11.7.1 Exporting Definitions

When building a large system in Interlisp, it is often the case that there are record definitions, macros and the like that are needed by several different system files when running, analyzing and compiling the source code of the system, but which are not needed for running the compiled code. By using the DECLARE: file package command with tag DONTCOPY (page 11.26), these definitions can be kept out of the compiled files, and hence out of the system constructed by loading the compiled files files into Interlisp. This saves loading time, space in the resulting system, and whatever other overhead might be incurred by keeping those definitions around, e.g., burden on the record package to consider more possibilities in translating record accesses, or conflicts between system record fields and user record fields.

However, if the implementor wants to debug or compile code in the resulting system, the definitions are needed. And even if the definitions *had* been copied to the compiled files, a similar problem arises if one wants to work on system code in a regular Interlisp environment where none of the system files had been loaded. One could mandate that any definition needed by more than one file in the system should reside on a distinguished file of definitions, to be loaded into any environment where the system files are worked on. Unfortunately, this would keep the definitions away from where they logically belong. The EXPORT mechanism is designed to solve this problem.

To use the mechanism, the implementor identifies any definitions needed by files other than the one in which the definitions reside, and wraps the corresponding file package commands in the EXPORT file package command (page 11.27). Thereafter, GATHEREXPORTS can be used to make a single file containing all the exports.

(GATHEREXPORTS *FROMFILES TOFILE FLG*) [Function]

FROMFILES is a list of files containing EXPORT commands. GATHEREXPORTS extracts all the exported commands from those files and produces a loadable file *TOFILE* containing them. If *FLG* = EVAL, the expressions are evaluated as they are gathered; i.e., the exports are effectively loaded into the current environment as well as being written to *TOFILE*.

(IMPORTFILE *FILE RETURNFLG*) [Function]

If *RETURNFLG* is NIL, this loads any exported definitions from *FILE* into the current environment. If *RETURNFLG* is T, this returns a list of the exported definitions (evaluable expressions) without actually evaluating them.

(CHECKIMPORTS *FILES NOASKFLG*) [Function]

Checks each of the files in *FILES* to see if any exists in a version newer than the one from which the exports in memory were taken (GATHEREXPORTS and IMPORTFILE note the creation dates of the files involved), or if any file in the list has not had its exports loaded at all. If there are any such files, the user is asked for permission to IMPORTFILE each such file. If *NOASKFLG* is non-NIL, IMPORTFILE is performed without asking.

For example, suppose file FOO contains records R1, R2, and R3, macros BAR and BAZ, and constants CON1 and CON2. If the definitions of R1, R2, BAR, and BAZ are needed by files other than FOO, then the file commands for FOO might contain the command

```
(DECLARE: EVAL@COMPILE DONTCOPY
   (EXPORT (RECORDS R1 R2)
           (MACROS BAR BAZ))
   (RECORDS R3)
   (CONSTANTS BAZ))
```

None of the commands inside this DECLARE: would appear on FOO's compiled file, but (GATHEREXPORTS '(FOO) 'MYEXPORTS) would copy the record definitions for R1 and R2 and the macro definitions for

BAR and BAZ to the file MYEXPORTS.

11.7.2 FileVars

In each of the file package commands described above, if the litatom * follows the command type,[7] the form following the *, i.e., CADDR of the command, is evaluated and its value used in executing the command, e.g., (FNS * (APPEND FNS1 FNS2)). When this form is a litatom, e.g. (FNS * FOOFNS), we say that the variable is a "filevar". Note that (COMS * *FORM*) provides a way of *computing* what should be done by MAKEFILE.

Example:

```
← (SETQ FOOFNS '(FOO1 FOO2 FOO3))
(FOO1 FOO2 FOO3)
← (SETQ FOOCOMS
     '( (FNS * FOOFNS)
        (VARS FIE)
        (PROP MACRO FOO1 FOO2)
        (P (MOVD 'FOO1 'FIE1))]
← (MAKEFILE 'FOO)
```

would create a file FOO containing:

```
(FILECREATED "time and date the file was made" . "other information")
(PRETTYCOMPRINT FOOCOMS)
(RPAQQ FOOCOMS ((FNS * FOOFNS) ···)
(RPAQQ FOOFNS (FOO1 FOO3 FOO3))
(DEFINEQ "definitions of FOO1, FOO2, and FOO3")
(RPAQQ FIE "value of FIE")
(PUTPROPS FOO1 MACRO PROPVALUE)
(PUTPROPS FOO2 MACRO PROPVALUE)
(MOVD (QUOTE FOO1) (QUOTE FIE1))
STOP
```

11.7.3 Defining New File Package Commands

A file package command is defined by specifying the values of certain properties. The user can specify the various attributes of a file package command for a new command, or respecify them for an existing command. The following properties are used:

MACRO Defines how to dump the file package command. Used by MAKEFILE. Value is a pair (*ARGS* . *COMS*). The "arguments" to the file package command are substituted for *ARGS* throughout *COMS*, and the result treated as a list of file package commands. For example, following (FILEPKGCOM 'FOO 'MACRO '((X Y) .

[7] Except for the PROP and IFPROP commands, in which case the * follows the property name, e.g., (PROP MACRO * FOOMACROS).

COMS)), the file package command (FOO A B) will cause A to be substituted for X and B for Y throughout *COMS*, and then *COMS* treated as a list of commands.

The substitution is carried out by SUBPAIR (page 2.24), so that the "argument list" for the macro can also be atomic. For example, if (X . *COMS*) was used instead of ((X Y) . *COMS*), then the command (FOO A B) would cause (A B) to be substituted for X throughout *COMS*.

Note: Filevars are evaluated *before* substitution. For example, if the litatom * follows *NAME* in the command, CADDR of the command is evaluated substituting in *COMS*.

ADD

Specifies how (if possible) to add an instance of an object of a particular type to a given file package command. Used by ADDTOFILE. Value is *FN*, a function of three arguments, *COM*, a file package command CAR of which is EQ to *COMMANDNAME*, *NAME*, a typed object, and *TYPE*, its type. *FN* should return T if it (undoably) adds *NAME* to *COM*, NIL if not. If no ADD property is specified, then the default is (1) if (CAR *COM*)=*TYPE* and (CADR *COM*)=*, and (CADDR *COM*) is a filevar (i.e. a literal atom), add *NAME* to the value of the filevar, or (2) if (CAR *COM*)=*TYPE* and (CADR *COM*) is not *, add *NAME* to (CDR *COM*).

Actually, the function is given a fourth argument, *NEAR*, which if non-NIL, means the function should try to add the item after *NEAR*. See discussion of ADDTOFILES?, page 11.8.

DELETE

Specifies how (if possible) to delete an instance of an object of a particular type from a given file package command. Used by DELFROMFILES. Value is *FN*, a function of three arguments, *COM*, *NAME*, and *TYPE*, same as for ADD. *FN* should return T if it (undoably) deletes *NAME* from *COM*, NIL if not. If no DELETE property is specified, then the default is (1) (CAR *COM*)=*TYPE* and (CADR *COM*)=*, and (CADDR *COM*) is a filevar (i.e. a literal atom), and *NAME* is contained in the value of the filevar, then remove *NAME* from the filevar, or (2) if (CAR *COM*)=*TYPE* and (CADR *COM*) is not *, and *NAME* is contained in (CDR *COM*), then remove *NAME* from (CDR *COM*).

If *FN* returns the value of ALL, it means that the command is now "empty", and can be deleted entirely from the command list.

CONTENTS
CONTAIN

Specifies whether an instance of an object of a given type is contained in a given file package command. Used by WHEREIS and INFILECOMS?. Value is *FN*, a function of three arguments, *COM*, a file package command CAR of which is EQ to *COMMANDNAME*, *NAME*, and *TYPE*. The interpretation of *NAME* is as follows: if *NAME* is NIL, *FN* should return a list of elements of type *TYPE* contained in *COM*. If *NAME* is T, *FN* should return T if there are *any* elements of type *TYPE* in *COM*. If *NAME* is an atom other than T or NIL, return T if *NAME* of type *TYPE* is contained in *COM*. Finally, if *NAME* is a list, return a list of those elements of type *TYPE* contained in *COM* that are also contained in *NAME*.

Note that it is sufficient for the CONTENTS function to simply return the list of items of type *TYPE* in command *COM*, i.e. it can in fact ignore the *NAME* argument. The *NAME* argument is supplied mainly for those situations where producing the entire list of items involves significantly more computation or creates more storage than simply determining whether a particular item (or any item) of type *TYPE* is contained in the command.

If a CONTENTS property is specified and the corresponding function application returns NIL and (CAR *COM*) = *TYPE*, then the operation indicated by *NAME* is performed (1) on the value of (CADDR *COM*), if (CADR *COM*) = *, otherwise (2) on (CDR *COM*). In other words, by specifying a CONTENTS property that returns NIL, e.g. the function NILL, the user specifies that a file package command of name FOO produces objects of file package type FOO and only objects of type FOO.

If the CONTENTS property is not provided, the command is simply expanded according to its MACRO definition, and each command on the resulting command list is then interrogated.

Note that if *COMMANDNAME* is a file package command that is used frequently, its expansion by the various parts of the system that need to interrogate files can result in a large number of CONSes and garbage collections. By informing the file package as to what this command actually does and does not produce via the CONTENTS property, this expansion is avoided. For example, suppose the user has a file package command called GRAMMARS which dumps various property lists but no functions. Thus, the file package could ignore this command when seeking information about FNS.

The function FILEPKGCOM is used to define new file package commands, or to change the attributes of existing commands. Note that it is possible to redefine the attributes of system file package commands, such as FNS or PROPS, and to cause unpredictable results.

(FILEPKGCOM *COMMANDNAME* *PROP*$_1$ *VAL*$_1$ ··· *PROP*$_N$ *VAL*$_N$) [NoSpread Function]

Nospread function for defining new file package commands, or changing attributes of existing file package commands. *PROP*$_i$ is one of of the property names described above; *VAL*$_i$ is the value to be given that property of the file package command *COMMANDNAME*. Returns *COMMANDNAME*.

(FILEPKGCOM *COMMANDNAME* *PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGTYPE *COMMANDNAME* returns an alist of all of the defined properties of *COMMANDNAME*, using the property names as keys.

11.8 FUNCTIONS FOR MANIPULATING FILE COMMAND LISTS

The following functions may be used to manipulate filecoms. Note that the argument *COMS* does *not* have to correspond to the filecoms for some file. For example, *COMS* can be the list of commands generated as a result of expanding a user defined file package command.

(INFILECOMS? *NAME* *TYPE* *COMS* —) [Function]

COMS is a list of file package commands, or a variable whose value is a list of file package commands. *TYPE* is a file package type. INFILECOMS? returns T if

NAME of type *TYPE* is "contained" in *COMS*.

If *NAME*=NIL, INFILECOMS? returns a list of all elements of type *TYPE*.

If *NAME*=T, INFILECOMS? returns T if there are *any* elements of type *TYPE* in *COMS*.

(ADDTOFILE *NAME TYPE FILE — —*) [Function]
Adds *NAME* of type *TYPE* to the file package commands for *FILE*. Uses ADDTOCOMS and MAKENEWCOM. Returns *FILE*. ADDTOFILE is undoable.

(DELFROMFILES *NAME TYPE FILES*) [Function]
Deletes all instances of *NAME* of type *TYPE* from the filecoms for each of the files on *FILES*. If *FILES* is a non-NIL litatom, (LIST *FILES*) is used. *FILES*=NIL defaults to FILELST. Returns a list of files from which *NAME* was actually removed. Uses DELFROMCOMS. DELFROMFILES is undoable.

Note: Deleting a function will also remove the function from any BLOCKS declarations in the filecoms.

(ADDTOCOMS *COMS NAME TYPE — —*) [Function]
Adds *NAME* as a *TYPE* to *COMS*, a list of file package commands or a variable whose value is a list of file package commands. Returns NIL if ADDTOCOMS was unable to find a command appropriate for adding *NAME* to *COMS*. ADDTOCOMS is undoable.

Note that the exact algorithm for adding commands depends the particular command itself. See discussion of the ADD property, in the description of FILEPKGCOM, page 11.32.

Note: ADDTOCOMS will not attempt to add an item to any command which is inside of a DECLARE: unless the user specified a specific name via the LISTNAME or NEAR option of ADDTOFILES?.

(DELFROMCOMS *COMS NAME TYPE*) [Function]
Deletes *NAME* as a *TYPE* from *COMS*. Returns NIL if DELFROMCOMS was unable to modify *COMS* to delete *NAME*. DELFROMCOMS is undoable.

(MAKENEWCOM *NAME TYPE — —*) [Function]
Returns a file package command for dumping *NAME* of type *TYPE*. Uses the procedure described in the discussion of NEWCOM, page 11.20.

(MOVETOFILE *TOFILE NAME TYPE FROMFILE*) [Function]
Moves the definition of *NAME* as a *TYPE* from *FROMFILE* to *TOFILE* by modifying the file commands in the appropriate way (with DELFROMFILES and ADDTOFILE).

Note that if *FROMFILE* is specified, the definition will be retrieved from that file, even if there is another definition currently in the user's environment.

(FILECOMSLST *FILE TYPE —*) [Function]
Returns a list of all objects of type *TYPE* in *FILE*.

TYPE can also be the name of a file package command. For example, (FILECOMSLST *FILE* 'BLOCKS) will return the list of all BLOCKS declaration in *FILE*. FILECOMSLST knows about expanding user defined file package commands.

(FILEFNSLST *FILE*) [Function]

Same as (FILECOMSLST *FILE* 'FNS).

(FILECOMS *FILE TYPE*) [Function]

Returns (PACK* *FILE* (OR *TYPE* 'COMS)). Note that (FILECOMS 'FOO) returns the litatom FOOCOMS, not the value of FOOCOMS.

(SMASHFILECOMS *FILE*) [Function]

Maps down (FILECOMSLST *FILE* 'FILEVARS) and sets to NOBIND all filevars (see page 11.30), i.e. any variable used in a command of the form (*COMMAND* * *VARIABLE*). Also sets (FILECOMS *FILE*) to NOBIND. Returns *FILE*.

11.9 SYMBOLIC FILE FORMAT

The file package manipulates symbolic files in a particular format. This format is defined so that the information in the file is easily readable when the file is listed, as well as being easily manipulated by the file package functions. In general, there is no reason for the user to manually change the contents of a symbolic file. However, in order to allow users to extend the file package, this section describes some of the functions used to write symbolic files, and other matters related to their format.

(PRETTYDEF *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*) [Function]

Writes a symbolic file in PRETTYPRINT format for loading, using FILERDTBL as its readtable. PRETTYDEF returns the name of the symbolic file that was created.

PRETTYDEF operates under a RESETLST (see page 9.19), so if an error occurs, or a control-D is typed, all files that PRETTYDEF has opened will be closed, the (partially complete) file being written will be deleted, and any undoable operations executed will be undone.[8]

PRTTYFNS is an optional list of function names. It is equivalent to including (FNS * *PRTTYFNS*) in the file package commands in *PRTTYCOMS*. *PRTTYFNS* is an anachronism from when PRETTYDEF did not use a list of file package commands, and should be specified as NIL.

PRTTYFILE is the name of the file on which the output is to be written. If *PRTTYFILE*=NIL, the primary output file is used. If *PRTTYFILE* is atomic the file is opened if not already open, and it becomes the primary output file. *PRTTYFILE* is closed at end of PRETTYDEF, and the primary output file is restored. Finally, if *PRTTYFILE* is a list, CAR of *PRTTYFILE* is assumed to be the file name, and is opened if not already open. In this case, the file is left open at end of PRETTYDEF.

[8] Since PRETTYDEF operates under a RESETLST, any RESETSAVEs executed in the file package commands will also be protected. For example, if one of the file package commands executes a (RESETSAVE (RADIX -8)), the RADIX will atomatically be restored.

PRTTYCOMS is a list of file package commands interpreted as described on page 11.21. If *PRTTYCOMS* is atomic, its top level value is used and an RPAQQ is written which will set that atom to the list of commands when the file is subsequently loaded. A PRETTYCOMPRINT expression (see below) will also be written which informs the user of the named atom or list of commands when the file is subsequently loaded.[9]

REPRINTFNS and *SOURCEFILE* are for use in conjunction with remaking a file (see page 11.10). *REPRINTFNS* can be a list of functions to be prettyprinted, or EXPRS, meaning prettyprint all functions with EXPR definitions, or ALL meaning prettyprint all functions either defined as EXPRs, or with EXPR properties. Note that doing a remake with *REPRINTFNS*=NIL makes sense if there have been changes in the file, but not to any of the functions, e.g., changes to variables or property lists. *SOURCEFILE* is the name of the file from which to copy the definitions for those functions that are *not* going to be prettyprinted, i.e., those not specified by *REPRINTFNS*. *SOURCEFILE*=T means to use most recent version (i.e., highest number) of *PRTTYFILE*, the second argument to PRETTYDEF. If *SOURCEFILE* cannot be found, PRETTYDEF prints the message "*FILE* NOT FOUND, SO IT WILL BE WRITTEN ANEW", and proceeds as it does when *REPRINTFNS* and *SOURCEFILE* are both NIL.

PRETTYDEF calls PRETTYPRINT with its second argument *PRETTYDEFLG*=T, so whenever PRETTYPRINT starts a new function, it prints (on the terminal) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Note that normally if PRETTYPRINT is given a litatom which is not defined as a function but is known to be on one of the files noticed by the file package, PRETTYPRINT will load in the definition (using LOADFNS) and print it. This is not done when PRETTYPRINT is called from PRETTYDEF.

(PRINTFNS *X* —) [Function]

X is a list of functions. PRINTFNS prettyprints a DEFINEQ epression that defines the functions to the primary output file using the primary readtable. Used by PRETTYDEF to implement the FNS file package command.

(PRINTDATE *FILE CHANGES*) [Function]

Prints the FILECREATED expression at beginning of PRETTYDEF files. *CHANGES* used by the file package.

(FILECREATED *X*) [NLambda NoSpread Function]

Prints a message (using LISPXPRINT) followed by the time and date the file was made, which is (CAR *X*). The message is the value of PRETTYHEADER, initially "FILE CREATED". If PRETTYHEADER=NIL, nothing is printed. (CDR *X*) contains information about the file, e.g., full name, address of file map, list of changed items, etc. FILECREATED also stores the time and date the file was made

[9] In addition, if any of the functions in the file are Nlambdas, PRETTYDEF will automatically print a DECLARE: expression suitable for informing the compiler about these functions, in case the user recompiles the file without having first loaded the nlambda functions. See page 12.6.

on the property list of the file under the property FILEDATES and performs other initialization for the file package.

(PRETTYCOMPRINT *x*) [NLambda Function]

Prints *x* (unevaluated) using LISPXPRINT, unless PRETTYHEADER=NIL.

PRETTYHEADER [Variable]

Value is the message printed by FILECREATED. PRETTYHEADER is initially "FILE CREATED". If PRETTYHEADER=NIL, neither FILECREATED nor PRETTYCOMPRINT will print anything. Thus, setting PRETTYHEADER to NIL will result in "silent loads". PRETTYHEADER is reset to NIL during greeting (page 14.5).

(FILECHANGES *FILE* *TYPE*) [Function]

Returns a list of the changed objects of file package type *TYPE* from the FILECREATED expression of *FILE*. If *TYPE*=NIL, returns an alist of all of the changes, with the file package types as the CARs of the elements..

(FILEDATE *FILE* —) [Function]

Returns the file date contained in the FILECREATED expression of *FILE*.

11.9.1 Copyright Notices

The system has a facility for automatically printing a copyright notice near the front of files, right after the FILECREATED expression, specifying the years it was edited and the copyright owner. The format of the copyright notice is:

```
(* Copyright (c) 1981 by Foo Bars Corporation)
```

Once a file has a copyright notice then every version will have a new copyright notice inserted into the file without user intervention. (The copyright information necessary to keep the copyright up to date is stored at the end of the file.).

Any year the file has been edited is considered a "copyright year" and therefore kept with the copyright information. For example, if a file has been edited in 1981, 1982, and 1984, then the copyright notice would look like:

```
(* Copyright (c) 1981,1982,1984 by Foo Bars Corporation)
```

When a file is made, if it has no copyright information, the system will ask the user to specify the copyright owner (if COPYRIGHTFLG=T). The user may specify one of the names from COPYRIGHTOWNERS, or give one of the following responses:

(1) Type a left-square-bracket. The system will then prompt for an arbitrary string which will be used as the owner-string

(2) Type a right-square-bracket, which specifies that the user really does not want a copyright notice.

(3) Type "NONE" which specifies that this file should never have a copyright notice.

For example, if COPYRIGHTOWNERS has the value

```
((BBN "Bolt Beranek and Newman Inc.")
 (XEROX "Xerox Corporation"))
```

then for a new file FOO the following interaction will take place:

```
Do you want to Copyright FOO? Yes
Copyright owner:   (user typed ?)
one of:
BBN - Bolt Beranek and Newman Inc.
XEROX - Xerox Corporation
NONE - no copyright ever for this file
[ - new copyright owner -- type one line of text
] - no copyright notice for this file now

Copyright owner: BBN
```

Then "Foo Bars Corporation" in the above copyright notice example would have been "Bolt Beranek and Newman Inc."

The following variables control the operation of the copyright facility:

COPYRIGHTFLG [Variable]

If COPYRIGHTFLG=NIL (default), the system will preserve old copyright information, but will not ask the user about copyrighting new files.

If COPYRIGHTFLG=T, then when a file is made, if it has no copyright information, the system will ask the user to specify the copyright owner.

If COPYRIGHTFLG=NEVER, the system will neither prompt for new copyright information nor preserve old copyright information.

COPYRIGHTOWNERS [Variable]

COPYRIGHTOWNERS is a list of entries of the form (*KEY OWNERSTRING*), where *KEY* is used as a response to ASKUSER and *OWNERSTRING* is a string which is the full identification of the owner.

DEFAULTCOPYRIGHTOWNER [Variable]

If the user does not respond in DWIMWAIT seconds to the copyright query, the value of DEFAULTCOPYRIGHTOWNER is used.

11.9.2 Functions Used Within Source Files

The following functions are normally only used within symbolic files, to set variable values, property values, etc. Most of these have special behavior depending on file package variables.

(RPAQ *VAR VALUE*) [NLambda Function]

An nlambda function like SETQ that sets the top level binding of *VAR* (unevaluated) to *VALUE*.

(RPAQQ *VAR VALUE*) [NLambda Function]

An nlambda function like SETQQ that sets the top level binding of *VAR* (unevaluated) to *VALUE* (unevaluated).

(RPAQ? *VAR* *VALUE*) [NLambda Function]

Similar to RPAQ, except that it does nothing if *VAR* already has a top level value other than NOBIND. Returns *VALUE* if *VAR* is reset, otherwise NIL.

RPAQ, RPAQQ, and RPAQ? generate errors if *x* is not a litatom. All are affected by the value of DFNFLG (page 5.9). If DFNFLG=ALLPROP (and the value of *VAR* is other than NOBIND), instead of setting *x*, the corresponding value is stored on the property list of *VAR* under the property VALUE. All are undoable.

(ADDTOVAR *VAR* x_1 x_2 ⋯ x_N) [NLambda NoSpread Function]

Each x_i that is not a member of the value of *VAR* is added to it, i.e. after ADDTOVAR completes, the value of *VAR* will be (UNION (LIST x_1 x_2 ⋯ x_N) *VAR*). ADDTOVAR is used by PRETTYDEF for implementing the ADDVARS command. It performs some file package related operations, i.e. "notices" that *VAR* has been changed. Returns the atom *VAR* (not the value of *VAR*).

(PUTPROPS *ATM* $PROP_1$ VAL_1 ⋯ $PROP_N$ VAL_N) [NLambda NoSpread Function]

Nlambda nospread version of PUTPROP (none of the arguments are evaluated). For $i=1 \cdots N$, puts property $PROP_i$, value VAL_i, on the property list of *ATM*. Performs some file package related operations, i.e., "notices" that the corresponding properties have been changed.

(SAVEPUT *ATM* *PROP* *VAL*) [Function]

Same as PUTPROP, but marks the corresponding property value as having been changed (used by the file package).

11.9.3 File Maps

A file map is a data structure which contains a symbolic 'map' of the contents of a file. Currently, this consists of the begin and end byte address (see GETFILEPTR, page 6.9) for each DEFINEQ expression in the file, the begin and end address for each function definition within the DEFINEQ, and the begin and end address for each compiled function.

MAKEFILE, PRETTYDEF, LOADFNS, RECOMPILE, and numerous other system functions depend heavily on the file map for efficient operation. For example, the file map enables LOADFNS to load selected function definitions simply by setting the file pointer to the corresponding address using SETFILEPTR, and then performing a single READ. Similarly, the file map is heavily used by the "remake" option of MAKEFILE (page 11.10): those function definitions that have been changed since the previous version are prettyprinted; the rest are simply copied from the old file to the new one, resulting in a considerable speedup.

Whenever a file is written by MAKEFILE, a file map for the new file is built. Building the map in this case essentially comes for free, since it requires only reading the current file pointer before and after each definition is written or copied. However, building the map does require that PRETTYPRINT *know* that it is printing a DEFINEQ expression. For this reason, the user should never print a DEFINEQ expression onto a file himself, but should instead always use the FNS file package command (page 11.22).

The file map is stored on the property list of the root name of the file, under the property FILEMAP. In addition, MAKEFILE writes the file map on the file itself. For cosmetic reasons, the file map is written as the last expression in the file. However, the *address* of the file map in the file is (over)written into the

FILECREATED expression that appears at the beginning of the file so that the file map can be rapidly accessed without having to scan the entire file. In most cases, LOAD and LOADFNS do not have to build the file map at all, since a file map will usually appear in the corresponding file, unless the file was written with BUILDMAPFLG=NIL, or was written outside of Interlisp.

Currently, file maps for *compiled* files are not written onto the files themselves. However, LOAD and LOADFNS will build maps for a compiled file when it is loaded, and store it on the property FILEMAP. Similary, LOADFNS will obtain and use the file map for a compiled file, when available.

The use and creation of file maps is controlled by the following variables:

BUILDMAPFLG [Variable]

Whenever a file is read by LOAD or LOADFNS, or written by MAKEFILE, a file map is automatically built unless BUILDMAPFLG=NIL. (BUILDMAPFLG is initially T.)

While building the map will not help the first reference to a file, it will help in future references. For example, if the user performs (LOADFROM 'FOO) where FOO does not contain a file map, the LOADFROM will be (slightly) slower than if FOO did contain a file map, but subsequent calls to LOADFNS for this version of FOO will be able to use the map that was built as the result of the LOADFROM, since it will be stored on FOO's FILEMAP property.

USEMAPFLG [Variable]

If USEMAPFLG=T (the initial setting), the functions that use file maps will first check the FILEMAP property to see if a file map for this file was previously obtained or built. If not, the first expression on the file is checked to see if it is a FILECREATED expression that also contains the address of a file map. If the file map is not on the FILEMAP property or in the file, a file map will be built (unless BUILDMAPFLG=NIL).

If USEMAPFLG=NIL, the FILEMAP property and the file will not be checked for the file map. This allows the user to recover in those cases where the file and its map for some reason do not agree. For example, if the user uses a text editor to change a symbolic file that contains a map (not recommended), inserting or deleting just one character will throw that map off. The functions which use file maps contain various integrity checks to enable them to detect that something is wrong, and to generate the error FILEMAP DOES NOT AGREE WITH CONTENTS OF *FILE*. In such cases, the user can set USEMAPFLG to NIL, causing the map contained in the file to be ignored, and then reexecute the operation.

File Maps

CHAPTER 12

THE COMPILER

The compiler is contained in the standard Interlisp system. It may be used to compile functions defined in the user's Interlisp system, or to compile definitions stored in a file. The resulting compiled code may be stored as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading.

The most common way to use the compiler is to use one of the file package functions, such as MAKEFILE (page 11.6), which automatically updates source files, and produces compiled versions. However, it is also possible to compile individual functions defined in the user's Interlisp system, by directly calling the compiler using functions such as COMPILE (page 12.10). No matter how the compiler is called, the function COMPSET is called which asks the user certain questions concerning the compilation. (COMPSET sets the free variables LAPFLG, STRF, SVFLG, LCFIL and LSTFIL which determine various modes of operation.) Those that can be answered "yes" or "no" can be answered with YES, Y, or T for "yes"; and NO, N, or NIL for "no". The questions are:

LISTING?

This asks whether to generate a listing of the compiled code. The LAP and machine code are usually not of interest but can be helpful in debugging macros. Possible answers are:

1	Prints output of pass 1, the LAP macro code.
2	Prints output of pass 2, the machine code.
YES	Prints output of both passes.
NO	Prints no listings.

The variable LAPFLG is set to the answer.

FILE:

This question (which only appears if the answer to LISTING? is affirmative) ask where the compiled code listing(s) should be written. Answering T will print the listings at the terminal. The variable LSTFIL is set to the answer.

REDEFINE?

This question asks whether the functions compiled should be redefined to their compiled definitions. If this is answered YES, the compiled code is stored and the function definition changed, otherwise the function definition remains unchanged.

The variable STRF is set to T (if this is answered YES) or NIL.

SAVE EXPRS?

This question asks whether the original defining EXPRs of functions should be saved. If answered YES, then before redefining a function to its compiled definition, the EXPR definition is saved on the property list of the function name. Otherwise they are discarded.

It is very useful to save the EXPR definitions, just in case the compiled function needs to be changed. The editing functions will retrieve this saved definition if it exists, rather than reading from a source file.

The variable SVFLG is set to T (if this is answered YES) or NIL.

OUTPUT FILE? This question asks whether (and where) the compiled definitions should be written into a file for later loading. If you answer with the name of a file, that file will be used. If you answer Y or YES, you will be asked the name of the file. If the file named is already open, it will continue to be used. If you answer T or TTY:, the output will be typed on the teletype (not particularly useful). If you answer N, NO, or NIL, output will *not* be done.

The variable LCFIL is set to the name of the file.

In order to make answering these questions easier, there are four other possible answers to the LISTING? question, which specify common compiling modes:

S Same as last setting. Uses the same answers to compiler questions as given for the last compilation.

F Compile to File, without redefining functions.

ST STore new definitions, saving EXPR definitions.

STF STore new definitions; Forget EXPR definitions.

Implicit in these answers are the answers to the questions on disposition of compiled code and EXPR definitions, so the questions REDEFINE? and SAVE EXPRS? would not be asked if these answers were given. OUTPUT FILE? would still be asked, however. For example:

```
←COMPILE((FACT FACT1 FACT2))
LISTING? ST
OUTPUT FILE? FACT.DCOM
(FACT COMPILING)
.
.
(FACT REDEFINED)
.
.
(FACT2 REDEFINED)
(FACT FACT1 FACT2)
←
```

This process caused the functions FACT, FACT1, and FACT2 to be compiled, redefined, and the compiled definitions also written on the file FACT.DCOM for subsequent loading.

12.1 COMPILER PRINTOUT

In Interlisp-D, for each function *FN* compiled, whether by TCOMPL, RECOMPILE, or COMPILE, the compiler prints:

```
(FN (ARG1 ··· ARGN) (uses: VAR1 ··· VARN) (calls: FN1 ··· FNN))
```

The message is printed at the beginning of the second pass of the compilation of *FN*. (*ARG*$_1$ ··· *ARG*$_N$) is the list of arguments to *FN*; following "uses:" are the free variables referenced or set in *FN* (not including global variables); following "calls:" are the undefined functions called within *FN*.

In Interlisp-10, for every function compiled, the compiler prints (*FN* (*ARG*$_1$ ··· *ARG*$_N$) (*FREE*$_1$ ··· *FREE*$_N$)), where *FREE*$_1$ ··· *FREE*$_N$ are the free variables referenced or set in *FN*.

If the compilation of *FN* causes the generation of one or more auxilary functions (see page 12.8), a compiler message will be printed for these functions before the message for *FN*, e.g.,

```
(FOOA0027 (X) (uses: XX))
(FOO (A B))
```

When compiling a block, the compiler first prints (*BLKNAME BLKFN*$_1$ *BLKFN*$_2$ ···). Then the normal message is printed for the entire block. The names of the arguments to the block are generated by suffixing "#" and a number to the block name, e.g., (FOOBLOCK (FOOBLOCK#0 FOOBLOCK#1) *FREE-VARIABLES*). Then a message is printed for each *entry* to the block.

In addition to the above output, both RECOMPILE and BRECOMPILE print the name of each function that is being copied from the old compiled file to the new compiled file. The normal compiler message is printed for each function that is actually compiled.

The compiler prints out error messages when it encounters problems compiling a function. For example:

```
----- In BAZ:
***** (BAZ - illegal RETURN)
-----
```

The above error message indicates that an "illegal RETURN" compiler error occurred while trying to compile the function BAZ. Some compiler errors cause the compilation to terminate, producing nothing; however, there are other compiler errors which do not stop compilation. The compiler error messages are described on page 12.20.

Compiler printout and error messages go to the file COUTFILE, initially T. COUTFILE can also be set to the name of a file opened for output, in which case all compiler printout will go to COUTFILE, i.e. the compiler will compile "silently." However, any error messages will be printed to both COUTFILE as well as T.

12.2 GLOBAL VARIABLES

Variables that appear on the list GLOBALVARS, or have the property GLOBALVAR with value T, or are declared with the GLOBALVARS file package command (page 11.25), are called global variables. Such variables are always accessed through their top level value when they are used freely in a compiled function. In other words, a reference to the value of this variable is equivalent to (GETTOPVAL (QUOTE *VARIABLE*)), regardless of whether or not it is bound in the current access chain. Similarly, (SETQ *VARIABLE VALUE*) will compile as (SETTOPVAL (QUOTE *VARIABLE*) *VALUE*).

All system parameters, unless otherwise specified, are declared as global variables. Thus, *rebinding* these variables in a deep bound system (like Interlisp-D) will not affect the behavior of the system: instead, the variables must be *reset* to their new values, and if they are to be restored to their original values, reset again. For example, the user might write

```
(SETQ GLOBALVARIABLE NEWVALUE)
FORM
(SETQ GLOBALVARIABLE OLDVALUE)
```

Note that in this case, if an error occurred during the evaluation of *FORM*, or a control-D was typed, the global variable would not be restored to its original value. The function RESETVAR (page 9.20) provides a convenient way of resetting global variables in such a way that their values are restored even if an error occurred or control-D is typed.

Note: Interlisp-10 employs a shallow binding scheme as described on page 7.1. There is no distinction between global variables and other types of variables: all variable references are to the variable's value cell. Thus, the cost of *accessing* a variable is small and independent of the depth of computation, whereas in a deep bound system, it can be expensive to search the stack for the most recent binding of a variable, hence the need for a mechanism like global variables. Note however that in a shallow bound system, the cost of rebinding a variable is somewhat higher than in a deep bound system (except when the variable is a LOCALVAR). For the purposes of compilation, global variables are treated the same as SPECVARS, i.e. their names are always visible on the stack when they are rebound.

12.3 LOCALVARS AND SPECVARS

In normal compiled and interpreted code, all variable bindings are accessible by lower level functions because the variable's name is associated with its value. We call such variables *special* variables, or specvars. As mentioned earlier, the block compiler normally does *not* associate names with variable values. Such unnamed variables are not accessible from outside the function which binds them and are therefore *local* to that function. We call such unnamed variables local variables, or localvars.

The time economies of local variables can be achieved without block compiling by use of declarations. Using local variables will increase the speed of compiled code; the price is the work of writing the necessary specvar declarations for those variables which need to be accessed from outside the block.

LOCALVARS and SPECVARS are variables that affect compilation. During regular compilation, SPECVARS is normally T, and LOCALVARS is NIL or a list. This configuration causes all variables bound in the functions being compiled to be treated as special *except* those that appear on LOCALVARS. During block compilation, LOCALVARS is normally T and SPECVARS is NIL or a list. All variables are then treated as local *except* those that appear on SPECVARS.

Declarations to set LOCALVARS and SPECVARS to other values, and therefore affect how variables are treated, may be used at several levels in the compilation process with varying scope.

(1) The declarations may be included in the filecoms of a file, by using the LOCALVARS and SPECVARS file package commands (page 11.25). The scope of the declaration is then the entire file:

```
··· (LOCALVARS . T) (SPECVARS X Y) ···
```

(2) The declarations may be included in block declarations; the scope is then the block, e.g.,

```
(BLOCKS ((FOOBLOCK FOO FIE (SPECVARS . T) (LOCALVARS X)))
```

(3) The declarations may also appear in individual functions, or in PROG's or LAMBDA's within a function, using the DECLARE function. In this case, the scope of the declaration is the function or the PROG or LAMBDA in which it appears. LOCALVARS and SPECVARS declarations must appear immediately after the variable list in the function, PROG, or LAMBDA, but intervening comments are permitted. For example:

```
(DEFINEQ ((FOO
         (LAMBDA (X Y)
             (DECLARE (LOCALVARS Y))
             (PROG (X Y Z)
                  (DECLARE (LOCALVARS X))
                  ... ]
```

If the above function is compiled (non-block), the outer X will be special, the X bound in the PROG will be local, and both bindings of Y will be local.

Declarations for LOCALVARS and SPECVARS can be used in two ways: either to cause variables to be treated the same whether the function(s) are block compiled or compiled normally, or to affect one compilation mode while not affecting the default in the other mode. For example:

```
(LAMBDA (X Y)
         (DECLARE (SPECVARS . T))
         (PROG (Z) ... ]
```

will cause X, Y, and Z to be specvars for both block and normal compilation while

```
(LAMBDA (X Y)
         (DECLARE (SPECVARS X))
         ... ]
```

will make X a specvar when block compiling, but when regular compiling the declaration will have no effect, because the default value of specvars would be T, and therefore *both* X and Y will be specvars by default.

Although LOCALVARS and SPECVARS declarations have the same form as other components of block declarations such as (LINKFNS . T), their operation is somewhat different because the two variables are not independent. (SPECVARS . T) will cause SPECVARS to be set to T, and LOCALVARS to be set to NIL. (SPECVARS V1 V2 ...) will have *no* effect if the value of SPECVARS is T, but if it is a list (or NIL), SPECVARS will be set to the union of its prior value and (V1 V2 ...). The operation of LOCALVARS is analogous. Thus, to affect both modes of compilation one of the two (LOCALVARS or SPECVARS) must be declared T before specifying a list for the other.

12.4 CONSTANTS

The function CONSTANT enables the user to define certain expressions as descriptions of their "constant" values. For example, if a user program needed a scratch list of length 30, the user could specify

(CONSTANT (to 30 collect NIL)) instead of (QUOTE (NIL NIL ···)). The former is more concise and displays the important parameter much more directly than the latter. CONSTANT can also be used to denote values that cannot be quoted directly, such as (CONSTANT (PACK NIL)), (CONSTANT (ARRAY 10)). It is also useful to parameterize quantities that are constant at run time but may differ at compile time, e.g. (CONSTANT BITSPERWORD) in a program is exactly equivalent to 36, if the variable BITSPERWORD is bound to 36 when the CONSTANT expression is evaluated at compile time.

When interpreted, the expression occuring as the argument to CONSTANT is evaluted each time it is encountered. If the CONSTANT form is compiled, however, the expression will be evaluated only once:

If the value of the expression has a readable print-name, then it will be evaluated at compile-time, and the value will be saved as a literal in the compiled function's definition, as if (QUOTE *VALUE-OF-EXPRESSION*) had appeared instead of (CONSTANT *EXPRESSION*).

If the value does not have a readable printname (e.g. the PACK and ARRAY examples above), then the expression itself will be saved with the function, and it will be evaluated when the function is first executed. The value will then be stored in the function's literals, and will be retrieved on future references.

Whereas the function CONSTANT attempts to evaluate the expression as soon as possible (compile-time, load-time, or first-run-time), the function DEFERREDCONSTANT will always defer the evaluation until first running. This is useful when the storage for the constant is excessive so that it shouldn't be allocated until (unless) the function is actually invoked.

Note: The function SELECTC (page 4.3) provides a mechanism for conparing a value to a number of constants.

(CONSTANTS *VAR*$_1$ *VAR*$_2$ ··· *VAR*$_N$) [NLambda NoSpread Function]

Defines *VAR*$_1$, ··· *VAR*$_N$ (unevaluated) to be compile-time constants. Whenever the compiler encounters a (free) reference to one of these constants, it will compile the form (CONSTANT *VAR*$_i$) instead.

If *VAR*$_i$ is a list of the form (*VAR FORM*), a free reference to the variable will compile as (CONSTANT *FORM*).

Constants can be saved using the CONSTANTS file package command (page 11.27).

12.5 COMPILING FUNCTION CALLS

When compiling the call to a function, the compiler must know the type of the function, to determine how the arguments should be prepared (evaluated/unevaluated, spread/nospread). There are three seperate cases: lambda, nlambda spread, and nlambda nospread functions.

To determine which of these three cases is appropriate, the compiler will first look for a definition among the functions in the file that is being compiled. The function can be defined anywhere in any of the files given as arguments to BCOMPL, TCOMPL, BRECOMPILE or RECOMPILE. If the function is not contained in the file, the compiler will look for other information in the variables NLAMA, NLAML, and LAMS, which can be set by the user:

NLAMA [Variable]

(for NLAMbda Atoms) A list of functions to be treated as nlambda nospread functions by the compiler.

NLAML [Variable]

(for NLAMbda List) A list of functions to be treated as nlambda spread functions by the compiler.

LAMS [Variable]

A list of functions to be treated as lambda functions by the compiler. Note that including functions on LAMS is only necessary to override in-core nlambda definitions, since in the absence of other information, the compiler assumes the function is a lambda.

If the function is not contained in a file, or on the lists NLAMA, NLAML, or LAMS, the compiler will look for a current definition in the Interlisp system, and use its type. If there is no current definition, next COMPILEUSERFN is called:

COMPILEUSERFN [Variable]

When compiling a function call, if the function type cannot be found by looking in files, the variables NLAMA, NLAML, or LAMS, or at a current definition, then if the value of COMPILEUSERFN is not NIL, the compiler calls (the value of) COMPILEUSERFN giving it as arguments CDR of the form and the form itself, i.e., the compiler does (APPLY* COMPILEUSERFN (CDR *FORM*) *FORM*). If a non-NIL value is returned, it is compiled instead of *FORM*. If NIL is returned, the compiler compiles the original expression as a call to a lambda spread that is not yet defined.

Note that COMPILEUSERFN is only called when the compiler encounters a *list* CAR of which is not the name of a defined function. The user can instruct the compiler about how to compile other data types via COMPILETYPELST, page 12.9.

CLISP uses COMPILEUSERFN to tell the compiler how to compile iterative statements, IF-THEN-ELSE statements, and pattern match constructs (See page 12.9).

If the compiler cannot determine the function type by any of the means above, it assumes that the function is a lambda function, and its arguments are to be evaluated. The function is also added to the value of ALAMS:

ALAMS [Variable]

(for Assumed LAMbdaS) A list of functions to that the compiler has assumed to be lambda functions. ALAMS is not used by the compiler; it is maintained for the user's benefit so that the user can check to see whether any incorrect assumptions were made.

If there are nlambda functions called from the functions being compiled, and they are only defined in a separate file, they must be included on NLAMA or NLAML, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. Note that this is only necessary if the compiler does not "know" about the function. If the function is defined at compile time, or is handled via a macro, or is contained in the same group of files as the functions that call it, the compiler will automatically handle calls to that function correctly.

12.6 FUNCTION AND FUNCTIONAL ARGUMENTS

Compiling the function FUNCTION (page 5.15) may involve creating and compiling a seperate "auxiliary function", which will be called at run time. An auxiliary function is named by attaching a GENSYM (page 2.11) to the end of the name of the function in which they appear, e.g., FOOA0003. For example, suppose FOO is defined as (LAMBDA (X) ··· (FOO1 X (FUNCTION ···)) ···) and compiled. When FOO is run, FOO1 will be called with two arguments, X, and FOOA000*N* and FOO1 will call FOOA000*N* each time it uses its functional argument.

Compiling FUNCTION will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (MAPCAR, MAPLIST, etc.). Note that a considerable savings in time could be achieved by making FOO1 compile open via a computed macro (page 5.17), e.g.

```
(Z (LIST (SUBST (CADADR Z)
                (QUOTE FN)
                DEF)
         (CAR Z)))
```

DEF is the definition of FOO1 as a function of just its first argument, and FN is the name used for its functional argument in its definition. In this case, (FOO1 X (FUNCTION ···)) would compile as an expression, containing the argument to FUNCTION as an open LAMBDA expression. Thus you save not only the function call to FOO1, but also each of the function calls to its functional argument. For example, if FOO1 operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time costs space, and the user must decide which is more important.

12.7 OPEN FUNCTIONS

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. If the amount of time spent *inside* the function is small, this function calling time will be a significant percentage of the total time required to use the function. Therefore, many "small" functions, e.g., CAR, CDR, EQ, NOT, CONS are always compiled "open", i.e., they do not result in a function call. Other larger functions such as PROG, SELECTQ, MAPC, etc. are compiled open because they are frequently used. The user can make other functions compile open via MACRO definitions (see page 5.17). The user can also affect the compiled code via COMPILEUSERFN (page 12.7) and COMPILETYPELST (page 12.9).

12.8 COMPILETYPELST

Most of the compiler's mechanism deals with how to handle forms (lists) and variables (literal atoms). The user can affect the compiler's behaviour with respect to lists and literal atoms in a number of ways, e.g. macros, declarations, COMPILEUSERFN (page 12.7), etc. COMPILETYPELST allows the user to tell the compiler what to do when it encounters a data type *other* than a list or an atom. It is the facility in the compiler that corresponds to DEFEVAL (page 5.11) for the interpreter.

COMPILETYPELST [Variable]

A list of elements of the form (*TYPENAME* . *FUNCTION*). Whenever the compiler encounters a datum that is not a list and not an atom (or a number) in a context where the datum is being evaluated, the type name of the datum is looked up on COMPILETYPELST. If an entry appears CAR of which is equal to the type name, CDR of that entry is applied to the datum. If the value returned by this application is *not* EQ to the datum, then that value is compiled instead. If the value *is* EQ to the datum, or if there is no entry on COMPILETYPELST for this type name, the compiler simply compiles the datum as (QUOTE *DATUM*).

12.9 COMPILING CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the definitions must first be DWIMIFYed (page 16.14). The user can automate this process in several ways:

(1) If the variable DWIMIFYCOMPFLG is T, the compiler will always DWIMIFY expressions before compiling them. DWIMIFYCOMPFLG is initially NIL.

(2) If a file has the property FILETYPE with value CLISP on its property list, TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE will operate as though DWIMIFYCOMPFLG is T and DWIMIFY all expressions before compiling.

(3) If the function definition has a local CLISP declaration (see page 16.10), including a null declaration, i.e., just (CLISP:), the definition will be automatically DWIMIFYed before compiling.

Note: COMPILEUSERFN (page 12.7) is defined to call DWIMIFY on iterative statements, IF-THEN statements, and fetch, replace, and match expressions, i.e., any CLISP construct which can be recognized by its CAR of form. Thus, if the only CLISP constructs in a function appear inside of iterative statements, IF statements, etc., the function does not have to be dwimified before compiling.

If DWIMIFY is ever unsuccessful in processing a CLISP expression, it will print the error message UNABLE TO DWIMIFY followed by the expression, and go into a break.[8] The user can exit the break in several different ways: (1) type OK to the break, which will cause the compiler to try again, e.g. the user could define some missing records while in the break, and then continue; or (2) type ↑, which will cause the compiler to simply compile the expression as is, i.e. as though CLISP had not been enabled in the first place; or (3) return an expression to be compiled in its place by using the RETURN break command (page 9.3).

Note: TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE all scan the entire file before doing any compiling, and take note of the names of all functions that are defined in the file as well as the names of all variables that are set by adding them to NOFIXFNSLST and NOFIXVARSLST, respectively. Thus,

[8] unless DWIMESSGAG=T. In this case, the expression is just compiled as is, i.e. as though clisp had not been enabled.

if a function is not currently defined, but *is* defined in the file being compiled, when DWIMIFY is called before compiling, it will not attempt to interpret the function name as CLISP when it appears as CAR of a form. DWIMIFY also takes into account variables that have been declared to be LOCALVARS, or SPECVARS, either via block declarations or DECLARE expressions in the function being compiled, and does not attempt spelling correction on these variables. The declaration USEDFREE may also be used to declare variables simply used freely in a function. These variables will also be left alone by DWIMIFY. Finally, NOSPELLFLG (page 15.12) is reset to T when compiling functions from a file (as opposed to from their in-core definition) so as to suppress spelling correction.

12.10 COMPILER FUNCTIONS

Normally, the compiler is envoked through file package commands that keep track of the state of functions, and manage a set of files, such as MAKEFILE (page 11.6). However, it is also possible to explicitly call the compiler using one of a number of functions. Functions may be compiled from in-core definitions (via COMPILE), or from definitions in files (TCOMPL), or from a combination of in-core and file definitions (RECOMPILE).

TCOMPL and RECOMPILE produce "compiled" files. Compiled files usually have the same name as the symbolic file they were made from, suffixed with DCOM (Interlisp-D) or COM (Interlisp-10).[9] The file name is constructed from the name field only, e.g., (TCOMPL '<BOBROW>FOO.TEM;3) produces FOO.DCOM on the connected directory. The version number will be the standard default.

A "compiled file" contains the same expressions as the original symbolic file, except that (1) a special FILECREATED expression appears at the front of the file which contains information used by the file package, and which causes the message COMPILED ON *DATE* to be printed when the file is loaded;[10] (2) every DEFINEQ in the symbolic file is replaced by the corresponding compiled definitions in the compiled file; and (3) expressions following a DONTCOPY tag inside of a DECLARE: (page 11.26) that appears in the symbolic file are not copied to the compiled file. The compiled definitions appear at the front of the compiled file, i.e., before the other expressions in the symbolic file, *regardless of where they appear in the symbolic file.* The only exceptions are expressions that follow a FIRST tag inside of a DECLARE: (page 11.26). This "compiled" file can be loaded into any Interlisp system with LOAD (page 11.4).

Note: When a function is compiled from its in-core definition (as opposed to being compiled from a definition in a file), and the function has been modified by BREAK, TRACE, BREAKIN, or ADVISE, it is first restored to its original state, and a message is printed out, e.g., FOO UNBROKEN. If the function is not defined as an EXPR, the value of the function's EXPR property is used for the compilation, if there is one. If there is no EXPR property, and the compilation is being performed by RECOMPILE, the definition of the function is obtained from the file (using LOADFNS). Otherwise, the compiler prints (*FN* NOT COMPILEABLE), and goes on to the next function.

(COMPILE *X* *FLG*) [Function]

X is a list of functions (if atomic, (LIST *X*) is used). COMPILE first asks the standard compiler questions, and then compiles each function on *X*, using its in-core definition. Returns *X*.

[9] The compiled file suffix is stored as the value of the variable COMPILE.EXT.

[10] The actual string printed is the value of COMPILEHEADER, initially "compiled on".

If compiled definitions are being written to a file, the file is closed unless *FLG*=T.

(COMPILE1 *FN DEF* —) [Function]

Compiles *DEF*, redefining *FN* if STRF=T (STRF is one of the variables set by COMPSET, page 12.1). COMPILE1 is used by COMPILE, TCOMPL, and RECOMPILE.

If DWIMIFYCOMPFLG is T, or *DEF* contains a CLISP declaration, *DEF* is dwimified before compiling. See page 12.9.

(TCOMPL *FILES*) [Function]

TCOMPL is used to "compile files"; given a symbolic LOAD file (e.g., one created by MAKEFILE), it produces a "compiled file". *FILES* is a list of symbolic files to be compiled (if atomic, (LIST *FILES*) is used). TCOMPL asks the standard compiler questions, except for "OUTPUT FILE:". The output from the compilation of each symbolic file is written on a file of the same name suffixed with DCOM, e.g., (TCOMPL '(SYM1 SYM2)) produces two files, SYM1.DCOM and SYM2.DCOM.

TCOMPL processes the files one at a time, reading in the entire file. For each FILECREATED expression, the list of functions that were marked as changed by the file package is noted, and the FILECREATED expression is written onto the output file. For each DEFINEQ expression, TCOMPL adds any nlambda functions defined in the DEFINEQ to NLAMA or NLAML, and adds lambda functions to LAMS, so that calls to these functions will be compiled correctly (see page 12.7).[11] Expressions beginning with DECLARE: are processed specially (see page 11.26). All other expressions are collected to be subsequently written onto the output file.

After processing the file in this fashion, TCOMPL compiles each function, except for those functions which appear on the list DONTCOMPILEFNS,[12] and writes the compiled definition onto the output file. TCOMPL then writes onto the output file the other expressions found in the symbolic file.

Note: If the rootname of a file has the property FILETYPE with value CLISP, or value a list containing CLISP, TCOMPL rebinds DWIMIFYCOMPFLG to T while compiling the functions on *FILE*, so the compiler will DWIMIFY all expressions before compiling them. See page 12.9.

TCOMPL returns a list of the names of the output files. All files are properly terminated and closed. If the compilation of any file is aborted via an error or control-D, all files are properly closed, and the (partially complete) compiled file is deleted.

(RECOMPILE *PFILE CFILE FNS*) [Function]

The purpose of RECOMPILE is to allow the user to update a compiled file without recompiling every function in the file. RECOMPILE does this by using the results of

[11] NLAMA, NLAML, and LAMS are rebound to their top level values (using RESETVAR) by TCOMPL, RECOMPILE, BCOMPL, BRECOMPILE, COMPILE, and BLOCKCOMPILE, so that any additions to these lists while inside of these functions will not propagate outside.

[12] Initially NIL. DONTCOMPILEFNS might be used for functions that compile open, since their definitions would be superfluous when operating with the compiled file. Note that DONTCOMPILEFNS can be set via block declarations (see page 12.14).

a previous compilation. It produces a compiled file similar to one that would have been produced by TCOMPL, but at a considerable savings in time by only compiling selected functions, and copying the compiled definitions for the remainder of the functions in the file from an earlier TCOMPL or RECOMPILE file.

PFILE is the name of the Pretty file (source file) to be compiled; *CFILE* is the name of the Compiled file containing compiled definitions that may be copied. *FNS* indicates which functions in *PFILE* are to be recompiled, e.g., have been changed or defined for the first time since *CFILE* was made. Note that *PFILE*, not *FNS*, drives RECOMPILE.

RECOMPILE asks the standard compiler questions, except for "OUTPUT FILE:". As with TCOMPL, the output automatically goes to *PFILE*.DCOM. RECOMPILE processes *PFILE* the same as does TCOMPL except that DEFINEQ expressions are not actually read into core. Instead, RECOMPILE uses the filemap (see page 11.38) to obtain a list of the functions contained in *PFILE*. The filemap enables RECOMPILE to skip over the DEFINEQs in the file by simply resetting the file pointer, so that in most cases the scan of the symbolic file is very fast (the only processing required is the reading of the non-DEFINEQs and the processing of the DECLARE: expressions as with TCOMPL). A map is built if the symbolic file does not already contain one, for example if it was written in an earlier system, or with BUILDMAPFLG=NIL (page 11.39).

After this initial scan of *PFILE*, RECOMPILE then processes the functions defined in the file. For each function in *PFILE*, RECOMPILE determines whether or not the function is to be (re)compiled. Functions that are members of DONTCOMPILEFNS are simply ignored. Otherwise, a function is recompiled if (1) *FNS* is a list and the function is a member of that list; or (2) *FNS*=T or EXPRS and the function is an EXPR; or (3) *FNS*=CHANGES and the function is marked as having been changed in the FILECREATED expression in *PFILE*; or (4) *FNS*=ALL.[13] If a function is not to be recompiled, RECOMPILE obtains its compiled definition from *CFILE*, and copies it (and all generated subfunctions) to the output file, *PFILE*.DCOM. If the function does not appear on *CFILE*, RECOMPILE simply recompiles it. Finally, after processing all functions, RECOMPILE writes out all other expressions that were collected in the prescan of *PFILE*.

If *CFILE*=NIL, *PFILE*.DCOM (the old version of the output file) is used for copying *from*. If both *FNS* and *CFILE* are NIL, *FNS* is set to the value of RECOMPILEDEFAULT, which is initially EXPRS. This is the most common usage. Typically, the functions the user has changed will have been UNSAVEDEFed by the editor, and therefore will be EXPRs. Thus the user can perform his edits, dump the file, and then simply (RECOMPILE '*FILE*) to update the compiled file.

The value of RECOMPILE is the new compiled file, *PFILE*.DCOM. If RECOMPILE is aborted due to an error or control-D, the new (partially complete) compiled file will be closed and deleted.

[13] If *FNS*=ALL, *CFILE* is superfluous, and does not have to be specified. This option may be used to compile a symbolic file that has never been compiled before, but which has already been loaded (since using TCOMPL would require reading the file in a second time).

RECOMPILE is designed to allow the user to conveniently and *efficiently* update a compiled file, even when the corresponding symbolic file has not been (completely) loaded. For example, the user can perform a LOADFROM (page 11.6) to "notice" a symbolic file, edit the functions he wants to change (the editor will automatically load those functions not already loaded), call MAKEFILE (page 11.6) to update the symbolic file (MAKEFILE will copy the unchanged functions from the old symbolic file), and then perform (RECOMPILE *PFILE*).

Note: Since PRETTYDEF automatically outputs a suitable DECLARE: expression to indicate which functions in the file (if any) are defined as NLAMBDAs, calls to these functions will be handled correctly, even though the NLAMBDA functions themselves may never be loaded, or even looked at, by RECOMPILE.

12.11 BLOCK COMPILING

Block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast. Thus, compiling a block consisting of just a single recursive function may be yield great savings if the function calls itself many times, e.g., EQUAL, COPY, and COUNT are block compiled in Interlisp-10.

The output of a block compilation is a single, usually large, function. Calls from within the block to functions outside of the block look like regular function calls, except that they are usually linked (see page 12.18). A block can be entered via several different functions, called entries.[14] These must be specified when the block is compiled. For example, the error block has three entries, ERRORX, INTERRUPT, and FAULT1. Similarly, the compiler block has nine entries.

Note: In Interlisp-D, block compiling is handled somewhat differently; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance improvement. Block compiling in Interlisp-D works by automatically renaming the block functions with special names, and calling these functions with the normal function-calling mechanisms. Specifically, a function *FN* is renamed to \BLOCK-NAME/FN. For example, function FOO in block BAR is renamed to "\BAR/FOO". Note that it is possible with this scheme to break functions internal to a block.

12.11.1 RETFNS

Another savings in block compilation arises from omitting most of the information on the stack about internal calls between functions in the block. However, if a function's name must be visible on the stack, e.g., if the function is to be returned from RETFROM, RETTO, RETEVAL, etc., it must be included on the list RETFNS.

[14] Actually the block is entered the same as every other function, i.e., at the top. However, the entry functions call the main block with their name as one of its arguments, and the block dispatches on the name, and jumps to the portion of the block corresponding to that entry point. The effect is thus the same as though there were several different entry points.

12.11.2 BLKAPPLYFNS

Normally, a call to APPLY from inside a block would be the same as a call to any other function outside of the block. If the first argument to APPLY turned out to be one of the entries to the block, the block would have to be reentered. BLKAPPLYFNS enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list BLKAPPLYFNS those functions which will be called in this fashion, and by using BLKAPPLY in place of APPLY, and BLKAPPLY* in place of APPLY*. If BLKAPPLY or BLKAPPLY* is given a function not on BLKAPPLYFNS, the effect is the same as a call to APPLY or APPLY* and no error is generated. Note however, that BLKAPPLYFNS must be set at *compile* time, not run time, and furthermore, that all functions on BLKAPPLYFNS must be in the block, or an error is generated (at compile time), NOT ON BLKFNS.

12.11.3 BLKLIBRARY

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since the user can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an EXPR definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list BLKLIBRARY, and their EXPR definitions on the property list of the functions under the property BLKLIBRARYDEF. When the block compiler compiles a form, it first checks to see if the function being called is one of the block functions. If not, and the function is on BLKLIBRARY, its definition is obtained from the property value of BLKLIBRARYDEF, and it is automatically included as part of the block. The functions ASSOC, EQUAL, GETPROP, LAST, LENGTH, LISPXWATCH, MEMB, MEMBER, NCONC1, NLEFT, NTH, /RPLNODE, and TAILP already have BLKLIBRARYDEF properties.

12.11.4 Block Declarations

Block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, linking, etc. To help with this, there is the BLOCKS file package command (page 11.25), which has the form:

(BLOCKS $BLOCK_1$ $BLOCK_2$ $\cdots$ $BLOCK_N$)

where each $BLOCK_i$ is a block declaration. The BLOCKS command outputs a DECLARE: expression, which is noticed by BCOMPL and BRECOMPILE. BCOMPL and BRECOMPILE are sensitive to these declarations and take the appropriate action.

Note: Masterscope includes a facility for checking the block declarations of a file or files for various anomalous conditions, e.g. functions in block declarations which aren't on the file(s), functions in ENTRIES not in the block, variables that may not need to be SPECVARS because they are not used freely below the places they are bound, etc. See page 13.1

The form of a block declaration is:

(*BLKNAME* $BLKFN_1$ ··· $BLKFN_M$ (VAR_1 . $VALUE_1$) ··· (VAR_N . $VALUE_N$))

BLKNAME is the name of a block. $BLKFN_1$ ··· $BLKFN_M$ are the functions in the block and correspond to *BLKFNS* in the call to BLOCKCOMPILE. The (VAR_i . $VALUE_i$) expressions indicate the settings for variables affecting the compilation of that block. If $VALUE_i$ is atomic, then VAR_i is set to $VALUE_i$ (e.g. (LINKFNS . T)), otherwise VAR_i is set to the UNION of $VALUE_i$ and the current value of the variable VAR_i. Also, expressions of the form (*VAR* * *FORM*) will cause *FORM* to be evaluated and the resulting list used as described above (e.g. (GLOBALVARS * MYGLOBALVARS)).

As an example, one of the block definitions for the editor is shown below. The block name is EDITBLOCK, it includes a number of functions (EDITL0, EDITL1, ··· EDITH), and it sets the variables ENTRIES, SPECVARS, RETFNS, GLOBALVARS, BLKAPPLYFNS, BLKLIBRARY, and NOLINKFNS:

```
(EDITBLOCK
   EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA EDITCOML
   EDITMAC EDITCOMS EDIT]UNDO UNDOEDITCOM UNDOEDITCOM1
   EDITSMASH EDITNCONC EDIT1F EDIT2F EDITNTH BPNT BPNT0
   BPNT1 RI RO LI LO BI BO EDITDEFAULT ## EDUP EDIT* EDOR
   EDRPT EDLOC EDLOCL EDIT: EDITMBD EDITXTR EDITELT
   EDITCONT EDITSW EDITMV EDITTO EDITBELOW EDITRAN TAILP
   EDITSAVE EDITH
   (ENTRIES EDITL0 ## UNDOEDITL)
   (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS **COMMENT**FLG
             PRETTYFLG UNDOLST UNDOLST1)
   (RETFNS EDITL0)
   (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS HISTORYCOMS
             EDITRACEFN)
   (BLKAPPLYFNS RI RO LI LO BI BO EDIT: EDITMBD EDITMV
             EDITXTR)
   (BLKLIBRARY LENGTH NTH LAST)
   (NOLINKFNS EDITRACEFN))
```

Whenever BCOMPL or BRECOMPILE encounter a block declaration, they rebind RETFNS, SPECVARS, GLOBALVARS, BLKLIBRARY, NOLINKFNS, LINKFNS, and DONTCOMPILEFNS to their top level values, bind BLKAPPLYFNS and ENTRIES to NIL, and bind BLKNAME to the first element of the declaration. They then scan the rest of the declaration, setting these variables as described above. When the declaration is exhausted, the block compiler is called and given BLKNAME, the list of block functions, and ENTRIES.

If a function appears in a block declaration, but is not defined in one of the files, then if it has an in-core definition, this definition is used and a message printed NOT ON FILE, COMPILING IN CORE DEFINITION. Otherwise, the message NOT COMPILEABLE, is printed and the block declaration processed as though the function were not on it, i.e. calls to the function will be compiled as external function calls.

Note that since all compiler variables are rebound for each block declaration, the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After finishing all blocks, BCOMPL and BRECOMPILE treat any functions in the file that did not appear in a block declaration in the same way as do TCOMPL and RECOMPILE. If the user wishes a function compiled separately as well as in a block, or if he wishes to compile some functions (not blockcompile), with some compiler variables changed, he can use a special pseudo-block declaration of the form

(NIL *BLKFN*$_1$ ··· *BLKFN*$_M$ (*VAR*$_1$. *VALUE*$_1$) ··· (*VAR*$_N$. *VALUE*$_N$))

which means that *BLKFN*$_1$ ··· *BLKFN*$_M$ should be compiled after first setting *VAR*$_1$ ··· *VAR*$_N$ as described above. For example,

```
(NIL CGETD FNTYP ARGLIST NARGS NCONC1 GENSYM (LINKFNS . T))
```

appearing as a "block declaration" will cause the six indicated functions to be compiled while LINKFNS=T so that all of their calls will be linked (except for those functions on NOLINKFNS).

12.11.5 Block Compiling Functions

There are three user level functions for block compiling, BLOCKCOMPILE, BCOMPL, and BRECOMPILE, corresponding to COMPILE, TCOMPL, and RECOMPILE. All of them ultimately call the same low level functions in the compiler, i.e., there is no "block compiler" per se. Instead, when block compiling, a flag is set to enable special treatment for SPECVARS, RETFNS, BLKAPPLYFNS, and for determining whether or not to link a function call. Note that all of the remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations, the user can intermix in a single file functions compiled normally, functions compiled normally with linked calls, and block compiled functions.

(BLOCKCOMPILE *BLKNAME BLKFNS ENTRIES FLG*) [Function]

BLKNAME is the name of a block, *BLKFNS* is a list of the functions comprising the block, and *ENTRIES* a list of entries to the block.

Each of the entries must also be on *BLKFNS* or an error is generated, NOT ON BLKFNS. If only one entry is specified, the block name can also be one of the *BLKFNS*, e.g., (BLOCKCOMPILE 'FOO '(FOO FIE FUM) '(FOO)). However, if more than one entry is specified, an error will be generated, CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME.

If *ENTRIES* is NIL, (LIST *BLKNAME*) is used, e.g., (BLOCKCOMPILE 'COUNT '(COUNT COUNT1))

If *BLKFNS* is NIL, (LIST *BLKNAME*) is used, e.g., (BLOCKCOMPILE 'EQUAL)

BLOCKCOMPILE asks the standard compiler questions and then begins compiling. As with COMPILE, if the compiled code is being written to a file, the file is closed unless *FLG*=T. The value of BLOCKCOMPILE is a list of the entries, or if *ENTRIES*=NIL, the value is *BLKNAME*.

The output of a call to BLOCKCOMPILE is one function definition for *BLKNAME*, plus definitions for each of the functions on *ENTRIES* if any. These entry functions are very short functions which immediately call *BLKNAME*.

(BCOMPL *FILES CFILE* — —) [Function]

FILES is a list of symbolic files (if atomic, (LIST *FILES*) is used). BCOMPL differs from TCOMPL in that it compiles all of the files at once, instead of one at a time, in order to permit one block to contain functions in several files. (If you have several files to be BCOMPLed *separately*, you must make several calls to BCOMPL.) Output is to *CFILE* if given, otherwise to a file whose name is (CAR *FILES*) suffixed with DCOM. For example, (BCOMPL '(EDIT WEDIT)) produces one file, EDIT.DCOM.

BCOMPL asks the standard compiler questions, except for "OUTPUT FILE:", then processes each file exactly the same as TCOMPL (page 12.11). BCOMPL next processes the block declarations as described above. Finally, it compiles those functions not mentioned in one of the block declarations, and then writes out all other expressions.

If *any* of the files have property FILETYPE with value CLISP, or a list containing CLISP, then DWIMIFYCOMPFLG is rebound to T for *all* of the files. See page 12.9.

The value of BCOMPL is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

Note that it is permissible to TCOMPL files set up for BCOMPL; the block declarations will simply have no effect. Similarly, you can BCOMPL a file that does not contain any block declarations and the result will be the same as having TCOMPLed it.

(BRECOMPILE *FILES CFILE FNS* —) [Function]

BRECOMPILE plays the same role for BCOMPL that RECOMPILE plays for TCOMPL. Its purpose is to allow the user to update a compiled file without requiring an entire BCOMPL.

FILES is a list of symbolic files (if atomic, (LIST *FILES*) is used). *CFILE* is the compiled file produced by BCOMPL or a previous BRECOMPILE that contains compiled definitions that may be copied. The interpretation of *FNS* is the same as with RECOMPILE.

BRECOMPILE asks the standard compiler questions except for "OUTPUT FILE:". As with BCOMPL, output automatically goes to *FILE*.DCOM, where *FILE* is the first file in *FILES*.

BRECOMPILE processes each file the same as RECOMPILE (page 12.11), then processes each block declaration. If *any* of the functions in the block are to be recompiled, the entire block must be (is) recompiled. Otherwise, the block is copied from *CFILE* as with RECOMPILE. For pseudo-block declarations of the form (NIL *FN1* ···), all variable assignments are made, but only those functions indicated by *FNS* are recompiled.

After completing the block declarations, BRECOMPILE processes all functions that do not appear in a block declaration, recompiling those dictated by *FNS*, and copying the compiled definitions of the remaining from *CFILE*.

Finally, BRECOMPILE writes onto the output file the "other expressions" collected in the initial scan of *FILES*.

The value of BRECOMPILE is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

If *CFILE*=NIL, the old version of *FILE*.DCOM is used, as with RECOMPILE. In addition, if *FNS* and *CFILE* are both NIL, *FNS* is set to the value of RECOMPILEDEFAULT, initially EXPRS.

12.12 LINKED FUNCTION CALLS

Note: Linked function calls are not implemented in Interlisp-D.

Conventional (non-linked) function calls from a compiled function go through the function definition cell, i.e., the definition of the called function is obtained from its function definition cell at call time. Thus, when the user breaks, advises, or otherwise modifies the definition of the function FOO, every function that subsequently calls it instead calls the modified function. For calls from the system functions, this is clearly *not* a desirable feature. For example, suppose that the user wishes to break on basic functions such as PRINT, EVAL, RPLACA, etc., which are used by the break package. We would like to guarantee that the system packages will survive through user modification (or destruction) of basic functions (unless the user specifically requests that the system packages also be modified). This protection is achieved by linked function calls.

For linked function calls, the definition of the called function is obtained at *link* time, i.e., when the calling function is defined, and stored in the literal table of the calling function. At *call* time, this definition is retrieved from where it was stored in the literal table, *not* from the function definition cell of the called function as it is for non-linked calls.

Note that while function calls from block compiled functions are *usually* linked (i.e. the default for blocks is to link),[15] and those from standardly compiled functions are *usually* non-linked, linking function calls and blockcompiling are independent features of the Interlisp compiler, i.e., linked function calls are possible, and frequently employed, from standardly compiled functions.

Note that normal function calls require only the called function's name in the literals of the compiled code, whereas a *linked* function call uses two literals and hence produces slightly larger compiled functions.

The compiler's decision as to whether to link a particular function call is determined by the variables LINKFNS and NOLINKFNS as follows:

(1) If the function appears on NOLINKFNS, the call is not linked;

[15] In Interlisp-10, linked function calls are actually a little slower and take more space than non-linked calls, so that the user might want to include (NOLINKFNS . T) in block declarations to override the default.

(2) If block compiling and the function is one of the block functions, the call is internal as described earlier;

(3) If the function appears on LINKFNS, the call is linked;

(4) If NOLINKFNS=T, the call is not linked;

(5) If block compiling, the call is linked;

(6) If LINKFNS=T, the call is linked;

(7) Otherwise the call is not linked.

Note that (1) takes precedence over (2), i.e., if a function appears on NOLINKFNS, the call to it is *not* linked, even if it is one of the functions in the block, i.e., the call will go outside of the block.

NOLINKFNS is initialized to various system functions such as ERRORSET, BREAK1, etc. LINKFNS is initialized to NIL. Thus if the user does not specify otherwise, all calls from a block compiled function (except for those to functions on NOLINKFNS) will be linked; all calls from standardly compiled functions will not be linked. However, when compiling system functions such as HELP, ERROR, ARGLIST, FNTYP, BREAK1, et al, LINKFNS is set to T so that even though these functions are not block compiled, all of their calls will be linked.

If a function is not defined at link time, i.e., when an attempt is made to link to it, it is linked instead to the function NOLINKDEF. When the function is later defined, the link can be completed by relinking the calling function using RELINK described below. Otherwise, if a function is run which attempts a linked call that was not completed, NOLINKDEF is called. If the function is now defined, i.e., it was defined at some point after the attempt was made to link to it, NOLINKDEF will quietly perform the link and continue the call. Otherwise, it will call FAULTAPPLY and proceed as described in page 15.6.

CALLS, BREAK on *FN1*-IN-*FN2* and ADVISE *FN1*-IN-*FN2* all work correctly for linked function calls, e.g., (BREAK '(FOO IN FIE)), where FOO is called from FIE via a linked function call. Note that control-H will *not* interrupt at linked function calls.

12.12.1 Relinking

The function RELINK is available for relinking a compiled function, i.e., updating all of its linked calls so that they use the definition extant at the time of the relink operation.

(RELINK *FN*) [Function]

FN is either the name of a function, a list of functions, an atom whose value is a list of functions, or the atom WORLD. RELINK performs the corresponding relinking operations. RELINK returns *FN*.

(RELINK 'WORLD) is possible because the compiled code reader maintains on LINKEDFNS a list of all user functions containing any linked calls. SYSLINKEDFNS is a list of all *system* functions that have any linked calls. (RELINK 'WORLD) performs both (RELINK LINKEDFNS) and (RELINK SYSLINKEDFNS).

Note: To relink a function in a block, one should relink the block, not the function.

It is important to stress that linking takes place when a function is *defined.* Thus, if FOO calls FIE via a linked call, and a bug is found in FIE, changing FIE is not sufficient; FOO must be relinked. Similarly, if FOO1, FOO2, and FOO3 are defined (in that order) in a file, and each call the others via linked calls, when a new version of the file is loaded, FOO1 will be linked to the *old* FOO2 and FOO3, since those definitions will be extant at the time it is read and defined. Similarly, FOO2 will link to the new FOO1 and *old* FOO3. Only FOO3 will link to the new FOO1 and FOO2. The user would have to perform (RELINK '(FOO1 FOO2 FOO3)) following the LOAD.

12.13 COMPILER ERROR MESSAGES

Messages describing errors in the function being compiled are also printed on the teletype. These messages are always preceded by *****. Unless otherwise indicated below, the compilation will continue.

(*FN* NOT ON FILE, COMPILING IN CORE DEFINITION)

From calls to BCOMPL and BRECOMPILE.

(*FN* NOT COMPILEABLE)

An EXPR definition for *FN* could not be found. In this case, no code is produced for *FN*, and the compiler proceeds to the next function to be compiled, if any.

(*FN* NOT FOUND)

Occurs when RECOMPILE or BRECOMPILE try to copy the compiled definition of *FN* from *CFILE*, and cannot find it. In this case, no code is copied and the compiler proceeds to the next function to be compiled, if any.

(*FN* NOT ON BLKFNS)

FN was specified as an entry to a block, or else was on BLKAPPLYFNS, but did not appear on the *BLKFNS*. In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(*FN* CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME)

In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(*BLKNAME* - USED BLKAPPLY WHEN NOT APPLICABLE)

BLKAPPLY is used in the block *BLKNAME*, but there are no BLKAPPLYFNS or ENTRIES declared for the block.

(*VAR* SHOULD BE A SPECVAR - USED FREELY BY *FN*)

In Interlisp-10, while compiling a block, the compiler has already generated code to bind *VAR* as a LOCALVAR, but now discovers that *FN* uses *VAR* freely. *VAR* should be declared a SPECVAR and the block recompiled.

((* --) COMMENT USED FOR VALUE)

A comment appears in a context where its value is being used, e.g. (LIST X (* --) Y). The compiled function will run, but the value at the point where the comment was used is "undefined."

((*FORM*) - NON-ATOMIC CAR OF FORM)

If user intended to treat the value of *FORM* as a function, he should use APPLY* (page 5.12). *FORM* is compiled as if APPLY* had been used.

((SETQ *VAR* *EXPR* --) BAD SETQ)

SETQ of more than two arguments.

(*FN* - USED AS ARG TO NUMBER FN?)

The value of a predicate, such as GREATERP or EQ, is used as an argument to a function that expects numbers, such as IPLUS.

(*FN* - NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT)

The compiler has assumed *FN* is the name of a function. If the user intended to treat the *value* of *FN* as a function, he must use APPLY* (page 5.12).

This message is printed when *FN* is not defined, and is also a local variable of the function being compiled. Note that earlier versions of the Interlisp-10 compiler did treat *FN* as a functional argument, and compiled code to evaluate it.

(*FN* - ILLEGAL RETURN)

RETURN encountered when not in PROG.

(*TG* - ILLEGAL GO)

GO encountered when not in a PROG.

(*TG* - MULTIPLY DEFINED TAG)

TG is a PROG label that is defined more than once in a single PROG. The second definition is ignored.

(*TG* - UNDEFINED TAG)

TG is a PROG label that is referenced but not defined in a PROG.

(*VAR* - NOT A BINDABLE VARIABLE)

VAR is NIL, T, or else not a literal atom.

(*VAR* *VAL* -- BAD PROG BINDING)

Occurs when there is a prog binding of the form (VAR VAL_1 $\cdots$ VAL_N).

(*TG* - MULTIPLY DEFINED TAG, ASSEMBLE)

TG is a label that is defined more than once in an assemble form.

(*TG* - UNDEFINED TAG, ASSEMBLE)

TG is a label that is referenced but not defined in an ASSEMBLE form.

(*TG* - MULTIPLY DEFINED TAG, LAP)

TG is a label that was encountered twice during the second pass of the compilation. If this error occurs with no indication of a multiply defined tag during pass one, the tag is in a LAP macro.

(*TG* - UNDEFINED TAG, LAP)

TG is a label that is referenced during the second pass of compilation and is not defined. LAP treats *TG* as though it were a COREVAL, and continues the compilation.

(*OP* - OPCODE? - ASSEMBLE)

OP appears as CAR of an assemble statement, and is illegal. See page 22.12 for legal assemble statements.

(NO BINARY CODE GENERATED OR LOADED FOR *FN*)

A previous error condition was sufficiently serious that binary code for *FN* cannot be loaded without causing an error.

CHAPTER 13

MASTERSCOPE

Masterscope is an interactive program for analyzing and cross referencing user programs. It contains facilities for analyzing user functions to determine what other functions are called, how and where variables are bound, set, or referenced, as well as which functions use particular record declarations. Masterscope is able to analyze definitions directly from a file as well as in-core definitions.

Masterscope maintains a database of the results of the analyses it performs. Via a simple command language, the user may interrogate the database, call the editor on those expressions in functions that were analyzed which use variables or functions in a particular way, or display the tree structure of function calls among any set of functions.

Masterscope is interfaced with the editor and file package so that when a function is edited or a new definition loaded in, Masterscope knows that it must re-analyze that function.

The following sample session illustrates some of these facilities.

```
←. ANALYZE FUNCTIONS ON RECORD                     [1]
................................                  [2]
NIL
←. WHO CALLS RECFIELDLOOK                         [3]
(RECFIELDLOOK ACCESSDEF ACCESSDEF2 EDITREC)
←. EDIT WHERE ANY CALL RECFIELDLOOK               [4]
RECFIELDLOOK :
(RECFIELDLOOK (CDR Y) FIELD)
tty:
*OK
ACCESSDEF :
(RECFIELDLOOK DECLST FIELD VAR1)
*OK
(RECFIELDLOOK USERRECLST FIELD)
*N VAR1
*OK
ACCESSDEF2 :
(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
tty:
(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
*N (CAR TAIL]
*OK
EDITREC :
(RECFIELDLOOK USERRECLST (CAR EDITRECX))
*OK
NIL
←. WHO CALLS ERROR                                [5]
..
```

```
(EDITREC)
←. SHOW PATHS TO RECFIELDLOOK FROM ACCESSDEF           [6]
(inverted tree)

1. RECFIELDLOOK RECFIELDLOOK
2.              ACCESSDEF
3.              ACCESSDEF2 ACCESSDEF2
4.                         ACCESSDEF
5.                         RECORDCHAIN ACCESSDEF
NIL
←. WHO CALLS WHO IN /FNS                               [7]
RECORDSTATEMENT --  /RPLNODE
RECORDECL1 --       /NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 --    /PUTHASH
UNCLISPTRAN --      /PUTHASH, /RPLNODE2
RECORDWORD --       /RPLACA
RECORD1 --          /RPLACA, /SETTOPVAL
EDITREC --          /SETTOPVAL
```

[1] The user directs that the functions on file RECORD be analyzed. The leading period and space specify that this line is a Masterscope command.[1]

[2] Masterscope prints a . whenever it (re)analyzes a function, to let the user know what it is happening.[2]

[3] The user asks which functions call RECFIELDLOOK. Masterscope responds with the list.

[4] The user asks to edit the expressions where the function RECFIELDLOOK is called. Masterscope calls EDITF on the functions it had analyzed that call RECFIELDLOOK, directing the editor to the appropriate expressions. The user then edits some of those expressions.[3]

[5] Next the user asks which functions call ERROR. Since some of the functions in the database have been changed, Masterscope re-analyzes the changed definitions (and prints out .'s for each function it analyzes). Masterscope responds that EDITREC is the only analyzed function that calls ERROR.

[6] The user asks to see a map of the ways in which RECFIELDLOOK is called from ACCESSDEF. A tree structure of the calls is displayed.

[1] The user may also call Masterscope directly by typing (MASTERSCOPE). Masterscope prints a greeting and prompts with "←. ". Within the top-level executive of Masterscope, the user may issue Masterscope commands, programmer's assistant commands, (e.g., REDO, FIX), or run programs. The user can exit from the Masterscope executive by typing OK. The function . is defined as a nlambda nospread function which interprets its argument as a Masterscope command, executes the command and returns.

[2] The feedback when Masterscope analyzes a function is controlled by the flag MSPRINTFLG: if MSPRINTFLG is the atom ".", Masterscope will print out a period. (If an error in the function is detected, "?" is printed instead.) If MSPRINTFLG is a number *N*, Masterscope will print the name of the function it is analyzing every *N*th function. If MSPRINTFLG is NIL, Masterscope won't print anything. Initial setting is ".". Note that the function name is printed when Masterscope starts analyzing, and the comma is printed when it finishes.

[3] In this example, the teletype editor is used. In Interlisp-D, if Dedit is enabled as the primary editor, it would be called to edit the appropriate functions (see page 20.1).

[7] The user then asks to see which functions call which functions in the list /FNS. Masterscope responds with a structured printout of these relations.

Below is a summary of the Masterscope commands, similar to what would be printed out by the HELP command (page 13.7). Optional elements are shown in brackets [ ]; alternatives are shown in braces {} separated with vertical bars | or are listed on separate lines; words in angle brackets <> are "meta-objects"; other lower-case words are "noise words" and may be omitted.

```
*------------------------------------------------------------------*
 a <command> is:

   [RE]ANALYZE <functions>
   ERASE <functions>
   show PATHS <pathoptions>
   <set> {<relation> | IS | ARE} <set>
   EDIT where <functions> [<relation> <set>] [ - <edit commands>]
   SHOW where <functions> <relation> <set>
   CHECK <files>
   FOR <variable> <set> <iterative statement tail>
-------------------------------------------------------------------
 a <set> is (at least one of):
 a determiner   + a type            + a specification

  THE           FUNCTIONS           [']{atom | list}
  ANY           VARIABLES           @ <predicate>
  WHICH         PROPERTY NAMES      IN <expression>
  WHO           RECORDS             <relation>ING <set>
                FIELDS              <relation>ED {BY | IN} <set>
                FILES               THAT <relation> <set>
                I.S.OPRS            LIKE <edit-pattern>
                                    ON <files>
                                    ON PATH <pathoptions>
                FIELDS OF <records>
                <blockword> {ON <files> | OF <functions>}
  <functions>, <files>, etc. are <set>s whose type is implied.
-------------------------------------------------------------------
 a <relation> is a verb and optional modifier:
  verbs:            modifiers (anywhere after the verb):
   CALL             {SOMEHOW | FOR EFFECT | FOR VALUE |
                                      DIRECTLY | INDIRECTLY}
   USE              AS a {RECORD | PROPERTY | record FIELD} name
   USE              AS a CLISP word
   USE              {FREELY | LOCALLY}
   SET              {FREELY | LOCALLY}
   SMASH            {FREELY | LOCALLY}
   TEST             {FREELY | LOCALLY}
   REFERENCE        {FREELY | LOCALLY}
   DECLARE          AS a {LOCALVAR | SPECVAR}
   BIND
   FETCH
   REPLACE      *-------------------------------------------------
  CREATE       | <blockword>:  ENTRIES, GLOBALVARS, FREEVARS,
  CONTAIN      |  SPECVARS, LOCALFREEVARS, BLKFNS or BLOCKFNS
-----------------------------------------------------------------
 <pathoptions>:              | abbreviations & synonyms:
                             |
   FROM <functions>          | FNS = FUNCTIONS  PROPS = PROPERTIES
   TO <functions>            | VARS = VARIABLES
   AVOIDING <functions>      | (& singular FN, VARIABLE, etc)
   NOTRACE <functions>       | FREE = FREELY    LOCAL = LOCALLY
   SEPARATE <functions>      | AMONG = AVOIDING NOT
   LINELENGTH <number>       |
-----------------------------------------------------------------
  <sets> may be joined by AND or OR or preceded by NOT.
  Any command can be followed by OUTPUT <filename>.
*---------------------------------------------------------------*
```

13.1 COMMAND LANGUAGE

The user communicates with Masterscope using an English-like command language, e.g., WHO CALLS PRINT. With these commands, the user can direct that functions be analyzed, interrogate Masterscope's database, and perform other operations. The commands deal with sets of functions, variables, etc., and relations between them (e.g., call, bind). Sets correspond to English nouns, relations to verbs.

A set of atoms can be specified in a variety of ways, either *explicitly*, e.g., FUNCTIONS ON FIE specifies the atoms in (FILEFNSLST 'FIE), or *implicitly*, e.g., NOT CALLING Y, where the meaning must be determined in the context of the rest of the command. Such sets of atoms are the basic building blocks which the command language deals with.

Masterscope also deals with relations *between* sets. For example, the relation CALL relates functions and other functions; the relations BIND and USE FREELY relate functions and variables. These relations are what get stored in the Masterscope database when functions are analyzed. In addition, Masterscope "knows" about file package conventions; CONTAIN relates files and various types of objects (functions, variables).

Sets and relations are used (along with a few additional words) to form sentence-like *commands*. For example, the command WHO ON 'FOO USE 'X FREELY will print out the list of functions contained in the file FOO which use the variable X freely. The command EDIT WHERE ANY CALLS 'ERROR will call EDITF on those functions which have previously been analyzed that directly call ERROR, pointing at each successive expression where the call to ERROR actually occurs.

13.1.1 Commands

The normal mode of communication with Masterscope is via "commands". These are sentences in the Masterscope command language which direct Masterscope to answer questions or perform various operations. The syntax of Masterscope commands is described below:

ANALYZE *SET* [Masterscope Command]

Analyze the functions in *SET* (and any functions called by them) and include the information gathered in the database. Masterscope will not re-analyzing a function if it thinks it already has valid information about that function in its database. The user may use the command REANALYZE (below) to force re-analysis.

Note that whenever a function is referred to in a command as a "subject" of one of the relations, it is automatically analyzed; the user need not give an explicit ANALYZE command. Thus, WHO IN MYFNS CALLS FIE will automatically analyze the functions in MYFNS if they have not already been analyzed.

Note also that only EXPR definitions will be analyzed; that is, Masterscope will not analyze compiled code. If there is no in-core definition for a function (either in the definition cell or an EXPR property), Masterscope will attempt to read in the definition from a file.[4] If necessary, the definition will be DWIMIFYed before analysis.

REANALYZE *SET* [Masterscope Command]

Causes Masterscope to reanalyze the functions in *SET* (and any functions called by them) even if it thinks it already has valid information in its database. For example, this would be necessary if the user had disabled or subverted the file package, e.g. performed PUTD's to change the definition of functions.

ERASE *SET* [Masterscope Command]

Erase all information about the functions in *SET* from the database. ERASE by itself clears the entire database.

SHOW PATHS *PATHOPTIONS* [Masterscope Command]

Displays a tree of function calls. *PATHOPTIONS* are described on page 13.14.

SET RELATION *SET* [Masterscope Command]
SET IS *SET* [Masterscope Command]
SET ARE *SET* [Masterscope Command]

This command has the same format as an English sentence with a subject (the first *SET*), a verb (the *RELATION* or IS or ARE), and an object (the second *SET*). Any of the *SET*s within the command may be preceded by the question determiners WHICH or WHO (or just WHO alone). For example, WHICH FUNCTIONS CALL X prints the list of functions that call the function X. *RELATION* may be one of the relation words in present tense (CALL, BIND, TEST, SMASH, etc.) or used as a passive (e.g., WHO IS CALLED BY WHO). Other variants are allowed, e.g. WHO DOES X CALL, IS FOO CALLED BY FIE, etc.

The interpretation of the command depends on the number of question elements present:

[4] Files which have been explicitly mentioned previously in some command are searched first. If the definition cannot be found on any of those files, Masterscope looks among the files on FILELST for a definition. If a function is found in this manner, Masterscope will print a message "(reading from *FILENAME*)". If no definition can be found at all, Masterscope will print a message "*FN* can't be analyzed". If the function previously was known, the message "*FN* disappeared!" is printed.

(1) If there is *no* question element, the command is treated as an assertion and Masterscope returns either T or NIL, depending on whether that assertion is true. Thus, ANY IN MYFNS CALL HELP will print T if any function in MYFNS call the function HELP, and NIL otherwise.

(2) If there is *one* question element, Masterscope returns the list of items for which the assertion would be true. For example MYFN BINDS WHO USED FREELY BY YOURFN prints the list of variables bound by MYFN which are also used freely by YOURFN.

(3) If there are two question elements, Masterscope will print a doubly indexed list:

```
←. WHO CALLS WHO IN /FNScr
RECORDSTATEMENT --  /RPLNODE
RECORDECL1 --       /NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 --    /PUTHASH
UNCLISPTRAN --      /PUTHASH, /RPLNODE2
RECORDWORD --       /RPLACA
RECORD1 --          /RPLACA, /SETTOPVAL
EDITREC --          /SETTOPVAL
```

EDIT WHERE *SET RELATION SET [- EDITCOMS]* [Masterscope Command]

(WHERE may be omitted.) The first *SET* refers to a set of functions. The EDIT command calls the editor on each expression where the *RELATION* actually occurs. For example, EDIT WHERE ANY CALL ERROR will call EDITF on each (analyzed) function which calls ERROR stopping within a TTY: at each call to ERROR. Currently one cannot EDIT WHERE a file which CONTAINS a datum, nor where one function CALLS another SOMEHOW.

EDITCOMS, if given, are a list of commands passed to EDITF to be performed at each expression. For example, EDIT WHERE ANY CALLS MYFN DIRECTLY - (SW 2 3) P will switch the first and second arguments to MYFN in every call to MYFN and print the result. EDIT WHERE ANY ON MYFILE CALL ANY NOT @ GETD will call the editor on any expression involving a call to an undefined function. Note that EDIT WHERE X SETS Y will point only at those expressions where Y is actually set, and will skip over places where Y is otherwise mentioned.

SHOW WHERE *SET RELATION SET* [Masterscope Command]

Like the EDIT command except merely prints out the expressions without calling the editor.

EDIT *SET [- EDITCOMS]* [Masterscope Command]

Calls EDITF on each function in *SET*. *EDITCOMS*, if given, will be passed as a list of editor commands to be executed. For example EDIT ANY CALLING FN1 - (R FN1 FN2) will replace FN1 by FN2 in those functions that call FN1.

DESCRIBE *SET* [Masterscope Command]

Prints out the BIND, USE FREELY and CALL information about the functions in *SET*. For example, the command DESCRIBE PRINTARGS might print out:

```
PRINTARGS[N,FLG]
```

```
binds:       TEM,LST,X
calls:       MSRECORDFILE,SPACES,PRIN1
called by:   PRINTSENTENCE,MSHELP,CHECKER
```

This shows that PRINTARGS has two arguments, N and FLG, binds internally the variables TEM, LST and X, calls MSRECORDFILE, SPACES and PRIN1 and is called by PRINTSENTENCE, MSHELP, and CHECKER.

The user can specify additional information to be included in the description. DESCRIBELST is a list each of whose elements is a list containing a descriptive string and a form. The form is evaluated (it can refer to the name of the funtion being described by the free variable FN); if it returns a non-NIL value, the description string is printed followed by the value. If the value is a list, its elements are printed with commas between them. For example, the entry ("types: " (GETRELATION FN '(USE TYPE) T) would include a listing of the types used by each function.

CHECK *SET* [Masterscope Command]

Checks for various anomolous conditions (mainly in the compiler declarations) for the files in *SET* (if *SET* is not given, FILELST is used). For example, this command will warn about variables which are bound but never referenced, functions in BLOCKS delarations which aren't on the file containing the declaration, functions declared as ENTRIES but not in the block, variables which may not need to be declared SPECVARS because they are not used freely below the places where they are bound, etc.

FOR *VARIABLE SET I.S.TAIL* [Masterscope Command]

This command provides a way of combining CLISP iterative statements with Masterscope. An iterative statement will be constructed in which *VARIABLE* is iteratively assigned to each element of *SET*, and then the iterative statement tail *I.S.TAIL* is executed. For example,

```
FOR X CALLED BY FOO WHEN CCODEP DO (PRINTOUT T X ,,, (ARGLIST
X) T)
```

will print out the name and argument list of all of the compiled functions which are called by FOO.

HELP [Masterscope Command]

Prints out a summary of Masterscope commands as shown on page 13.3. Optional elements are shown in brackets []; alternatives are shown in braces {} separated with vertical bars | or are listed on separate lines; words in angle brackets <> are "meta-objects"; other lower-case words are "noise words" and may be omitted.

Note: any command may be followed by OUTPUT *FILENAME* to send output to the given file rather than the terminal, e.g. WHO CALLS WHO OUTPUT CROSSREF.

13.1.2 Relations

A relation is specified by one of the keywords below. Some of these "verbs" accept modifiers. For example, USE, SET, SMASH and REFERENCE all may be modified by FREELY. The modifier may occur anywhere within the command.[5] Verbs can occur in the present tense (e.g., USE, CALLS, BINDS, USES) or as present or past participles (e.g., CALLING, BOUND, TESTED). The relations (with their modifiers) recognized by Masterscope are:

CALL [Masterscope Relation]

Function F1 calls F2 if the definition of F1 contains a form (F2 --), (APPLY (QUOTE F2) --), (FUNCTION F2), etc.

CALL SOMEHOW [Masterscope Relation]

One function calls another SOMEHOW if there is some path from the first to the other. That is, if F1 calls F2, and F2 calls F3, then F1 CALLS F3 SOMEHOW.

This information is not stored directly in the database; instead, Masterscope stores only information about direct function calls, and (re)computes the CALL SOMEHOW relation as necessary.

USE [Masterscope Relation]

If unmodified, the relation USE denotes variable usage in any way; it is the union of the relations SET, SMASH, TEST, and REFERENCE.

SET [Masterscope Relation]

A function SETs a variable if the function contains a form (SETQ var --), (SETQQ var --), etc.

SMASH [Masterscope Relation]

A function SMASHes a variable if the function calls a destructive list operation (RPLACA, RPLACD, DREMOVE, SORT, etc.) on the value of that variable. Masterscope will also find instances where the operation is performed on a "part" of the value of the variable; for example, if a function contains a form (RPLACA (NTH X 3) T) it will be noted as SMASHING X.

Note that if the function contains a sequence (SETQ Y X), (RPLACA Y T) then Y is noted as being smashed, but not X.

TEST [Masterscope Relation]

A variable is TESTed by a function if its value is only distinguished between NIL and non-NIL. For example, the form (COND ((AND X --) --)) *tests* the value of X.

REFERENCE [Masterscope Relation]

This relation includes all variable usage *except* for SET.

The verbs USE, SET, SMASH, TEST and REFERENCE may be modified by the words FREELY or LOCALLY. A variable is used FREELY if it is not bound in the function at the place of its use; alternatively, it is used LOCALLY if the use occurs within a PROG or LAMBDA that binds the variable.

[5] If there is more than one verb, any modifier *between* two verbs is assumed to modify the first one. For example, in USING ANY FREELY OR SETTING X, the FREELY modifies USING but not SETTING – the entire phrase is interpreted as the set of all functions which either use any variable freely or set the variable X, whether or not X is set freely.

Masterscope also distinguishes between CALL DIRECTLY and CALL INDIRECTLY. A function is called DIRECTLY if it occurs as CAR-of-form in a normal evaluation context. A function is called INDIRECTLY if its name appears in a context which does not imply its *immediate* evaluation, for example (SETQ Y (LIST (FUNCTION FOO) 3)).[6] In addition, CALL FOR EFFECT (where the value of the function is not used) is distinguished from CALL FOR VALUE.

BIND [Masterscope Relation]

The BIND relation between functions and variables includes both variables bound as function arguments and those bound in an internal PROG or LAMBDA expression.

USE AS A FIELD [Masterscope Relation]

Masterscope notes all uses of record field names within FETCH, REPLACE or CREATE expressions.

FETCH [Masterscope Relation]

Use of a field within a FETCH expression.

REPLACE [Masterscope Relation]

Use of a record field name within a REPLACE or CREATE expression.

USE AS A RECORD [Masterscope Relation]

Masterscope notes all uses of record names within CREATE or TYPE? expressions.[7]

CREATE [Masterscope Relation]

Use of a record name within a CREATE expression.

USE AS A PROPERTY NAME [Masterscope Relation]

Masterscope notes the property names used in GETPROP, PUTPROP, GETLIS, etc. expressions if the name is quoted. E.g. if a function contains a form (GETPROP X (QUOTE INTERP)), then that function USEs INTERP as a property name.

USE AS A CLISP WORD [Masterscope Relation]

Masterscope notes all iterative statement operators and user defined CLISP words as being used as a CLISP word.

CONTAIN [Masterscope Relation]

Files *contain* functions, records, and variables. This relation is not stored in the database but is computed using the file package.

DECLARE AS LOCALVAR [Masterscope Relation]
DECLARE AS SPECVAR [Masterscope Relation]

Masterscope notes internal "calls" to DECLARE from within functions.

The following abbreviations are recognized: FREE=FREELY, LOCAL=LOCALLY, PROP=PROPERTY, REF=REFERENCE. Also, the words A, AN and NAME (after AS) are "noise" words and may be omitted.

[6] The distinction is whether or not the compiled code of the caller would contain a direct call to the callee. Note that an occurrence of (FUNCTION FOO) as the functional argument to one of the built-in mapping functions which compile open is considered to be a direct call.

[7] Additionally, in X:FOO.FIE, FOO is used as a record name.

Note: Masterscope uses "templates" (page 13.16) to decide which relations hold between functions and their arguments. For example, the information that SORT SMASHes its first argument is contained in the template for SORT. Masterscope initially contains templates for most system functions which set variables, test their arguments, or perform destructive operations. The user may change existing templates or insert new ones in Masterscope's tables via the SETTEMPLATE function (page 13.19).

13.1.3 Sets

A "set" is a collection of things (functions, variables, etc.). A set is specified by a set phrase, consisting of a *determiner* (e.g., ANY, WHICH, WHO) followed by a *type* (e.g., FUNCTIONS, VARIABLES) followed by a *specification* (e.g., IN MYFNS, @ SUBRP). The determiner, type and specification may be used alone or in combination. For example, ANY FUNCTIONS IN MYFNS, ANY @ SUBRP, VARIABLES IN GLOBALVARS, and WHO are all acceptable set phrases. Set specifications, types and determiners are explained below:

13.1.3.1 Set Specifications

'*ATOM* [Masterscope Set Specification]

The simplest way to specify a set consisting of a single thing is by the name of that thing. For example, in the command WHO CALLS 'ERROR, the function ERROR is referred to by its name. Although the ' can be left out, to resolve possible ambiguities names should usually be quoted; e.g., WHO CALLS 'CALLS will return the list of functions which call the function CALLS.

'*LIST* [Masterscope Set Specification]

Sets consisting of several atoms may be specified by naming the atoms. For example, the command WHO USES '(A B) returns the list of functions that use the variables A or B.

IN *EXPRESSION* [Masterscope Set Specification]

The form *EXPRESSION* is evaluated, and its value is treated as a list of the elements of a set. For example, IN GLOBALVARS specifies the list of variables in the value of the variable GLOBALVARS.

@ *PREDICATE* [Masterscope Set Specification]

A set may also be specified by giving a predicate which the elements of that set must satisfy. *PREDICATE* is either a function name, a LAMBDA expression, or an expression in terms of the variable X. The specification @ *PREDICATE* represents all atom for which the value of *PREDICATE* is non-NIL. For example, @ EXPRP specifies all those atoms which have EXPR defintions; @ (STRPOSL X CLISPCHARRAY) specifies those atoms which contain CLISP characters. The universe to be searched is either determined by the context within the command (e.g., in WHO IN FOOFNS CALLS ANY NOT @ GETD, the predicate is only applied to functions which are called by any functions in the list FOOFNS), or in the extreme case, the universe defaults to the entire set of things which have been noticed by Masterscope, as in the command WHO IS @ EXPRP.

LIKE *ATOM* [Masterscope Set Specification]

ATOM may contain ESCs; it is used as a pattern to be matched (as in the editor).

For example, WHO LIKE /R$ IS CALLED BY ANY would find both /RPLACA and /RPLNODE.

A set may also be specified by giving a relation its members must have with the members of another set:

*RELATION*ING *SET* [Masterscope Set Specification]

*RELATION*ING is used here generically to mean any of the relation words in the present participle form (possibly with a modifier), e.g., USING, SETTING, CALLING, BINDING. *RELATION*ING *SET* specifies the set of all objects which have that relation with some element of *SET*. For example, CALLING X specifies the set of functions which call the function X; USING ANY IN FOOVARS FREELY specifies the set of functions which uses freely any variable in the value of FOOVARS.

*RELATION*ED BY *SET* [Masterscope Set Specification]

*RELATION*ED IN *SET* [Masterscope Set Specification]

This is similar to the *RELATION*ING construction. For example, CALLED BY ANY IN FOOFNS represents the set of functions which are called by any element of FOOFNS; USED FREELY BY ANY CALLING ERROR is the set of variables which are used freely by any function which also calls the function ERROR.

BLOCKTYPE OF *FUNCTIONS* [Masterscope Set Specification]

BLOCKTYPE ON *FILES* [Masterscope Set Specification]

These phrases allow the user to ask about BLOCKS declarations on files (see page 12.14). *BLOCKTYPE* is one of LOCALVARS, SPECVARS, GLOBALVARS, ENTRIES, BLKFNS, BLKAPPLYFNS, or RETFNS.

BLOCKTYPE OF *FUNCTIONS* specifies the names which are declared to be *BLOCKTYPE* in any blocks declaration which contain any of *FUNCTIONS* (a "set" of functions). The "functions" in *FUNCTIONS* can either be block names or just functions in a block. For example, WHICH ENTRIES OF ANY CALLING 'Y BIND ANY GLOBALVARS ON 'FOO.

BLOCKTYPE ON *FILES* specifies all names which are declared to be *BLOCKTYPE* on any of the given *FILES* (a "set" of files).

FIELDS OF *SET* [Masterscope Set Specification]

SET is a set of records. This denotes the field names of those records. For example, the command WHO USES ANY FIELDS OF BRECORD returns the list of all functions which do a fetch or replace with any of the field names declared in the record declaration of BRECORD.

KNOWN [Masterscope Set Specification]

The set of all functions which have been analyzed. For example, the command WHO IS KNOWN will print out the list of functions which have been analyzed.

THOSE [Masterscope Set Specification]

The set of things printed out by the last Masterscope question. For example, following the command WHO IS USED FREELY BY PARSE, the user could ask WHO BINDS THOSE to find out where those variables are bound.

ON PATH *PATHOPTIONS* [Masterscope Set Specification]

Refers to the set of functions which *would be* printed by the command SHOW PATHS

PATHOPTIONS. For example, IS FOO BOUND BY ANY ON PATH TO 'PARSE tests if FOO might be bound "above" the function PARSE. *PATHOPTIONS* are explained in detail on page 13.14.

Note: sets may also be specified with "relative clauses" introduced by the word THAT, e.g. THE FUNCTIONS THAT BIND 'X.

13.1.3.2 Set Determiners

Set phrases may be preceded by a *determiner.* A determiner is one of the words THE, ANY, WHO or WHICH. The "question" determiners (WHO and WHICH) are only meaningful in some of the commands, namely those that take the form of questions. ANY and WHO (or WHOM) can be used alone; they are "wild-card" elements, e.g., the command WHO USES ANY FREELY, will print out the names of all (known) functions which use any variable freely. If the determiner is omitted, ANY is assumed; e.g. the command WHO CALLS '(PRINT PRIN1 PRIN2) will print the list of functions which call *any* of PRINT, PRIN1, PRIN2. THE is also allowed, e.g. WHO USES THE RECORD FIELD FIELDX.

13.1.3.3 Set Types

Any set phrase has a *type*; that is, a set may specify either functions, variables, files, record names, record field names or property names. The type may be determined by the context within the command (e.g., in CALLED BY ANY ON FOO, the set ANY ON FOO is interpreted as meaning the *functions* on FOO since only functions can be CALLED), or the type may be given explicitly by the user (e.g., FUNCTIONS ON FIE). The following types are recognized: FUNCTIONS, VARIABLES, FILES, PROPERTY NAMES, RECORDS, FIELDS, I.S.OPRS.[8]

The type is used by Masterscope in a variety of ways when interpreting the set phrase:

(1) Set types are used to disambiguate possible parsings. For example, both commands WHO SETS ANY BOUND IN X OR USED BY Y and WHO SETS ANY BOUND IN X OR CALLED BY Y have the same general form. However, the first case is parsed as WHO SETS ANY (BOUND BY X OR USED BY Y) since both BOUND BY X and USED BY Y refer to variables; while the second case as WHO SETS ANY BOUND IN (X OR CALLED BY Y), since CALLED BY Y and X must refer to functions. Note that parentheses may be used to group phrases.

(2) The type is used to determine the modifier for USE: FOO USES WHICH RECORDS is equivalent to FOO USES WHO AS A RECORD FIELD.

(3) The interpretation of CONTAIN depends on the type of its object: the command WHAT FUNCTIONS ARE CONTAINED IN MYFILE prints the list of functions in MYFILE; WHAT RECORDS ARE ON MYFILE prints the list of records.

(4) The implicit "universe" in which a set expression is interpreted depends on the type: ANY VARIABLES @ GETD is interpreted as the set of all variables which have been noticed by Masterscope (i.e., bound or

[8] or abbreviations FNS, VARS, PROPNAMES or the singular forms FUNCTION, FN, VARIABLE, VAR, FILE, PROPNAME, RECORD, FIELD. Note that most of these types correspond to built-in "file package types" (see page 11.14).

used in any function which has been analyzed) that also have a definition. ANY FUNCTIONS @ (NEQ (GETTOPVAL X) 'NOBIND) is interpreted as the set of all functions which have been noticed (either analyzed or called by a function which has been analyzed) that also have a top-level value.

13.1.4 Conjunctions

Sets may be joined by the conjunctions AND and OR or preceded by NOT to form new sets. AND is always interpreted as meaning "intersection"; OR as "union", while NOT means "complement". For example, the set CALLING X AND NOT CALLED BY Y specifies the set of all functions which call the function X but are not called by Y.

Masterscope's interpretation of AND and OR follow LISP conventions rather than the conventional English interpretation. For example "calling X and Y" would, in English, be interpreted as the intersection of (CALLING X) and (CALLING Y); but Masterscope interprets CALLING X AND Y as CALLING ('X AND 'Y); which is the null set. Only sets may be joined with conjunctions: joining modifiers, as in USING X AS A RECORD FIELD OR PROPERTY NAME, is not allowed; in this case, the user must say USING X AS A RECORD FIELD OR USING X AS A PROPERTY NAME.

As described above, the type of sets is used to disambiguate parsings. The algorithm used is to first try to match the type of the phrases being joined and then try to join with the longest preceding phrase. In any case, the user may group phrases with parentheses to specify the manner in which conjunctions should be parsed.

13.2 PATHS

In trying to work with large programs, the user can lose track of the hierarchy of functions. The Masterscope SHOW PATHS command aids the user by providing a map showing the calling structure of a set of functions. SHOW PATHS prints out a tree structure showing which functions call which other functions. For example, the command SHOW PATHS FROM MSPARSE will print out the structure of Masterscope's parser:

```
1.MSPARSE  MSINIT MSMARKINVALID
2.         |      MSINITH MSINITH
3.         MSINTERPRET MSRECORDFILE
4.         |           MSPRINTWORDS
5.         |           PARSECOMMAND GETNEXTWORD CHECKADV
6.         |           |            PARSERELATION {a}
7.         |           |            PARSESET {b}
8.         |           |            PARSEOPTIONS {c}
9.         |           |            MERGECONJ GETNEXTWORD {5}
10.        |           GETNEXTWORD {5}
11.        |           FIXUPTYPES SUBJTYPE
12.        |           |          OBJTYPE
13.        |           FIXUPCONJUNCTIONS MERGECONJ {9}
14.        |                             MATCHSCORE
15.        MSPRINTSENTENCE
------------------------------------------------------ overflow - a
```

```
16.PARSERELATION GETNEXTWORD {5}
17.              CHECKADV
--------------------------------------------------------- overflow - b
19.PARSESET PARSESET
20.         GETNEXTWORD| {5}
21.         PARSERELATION {6}
22.         SUBPARSE GETNEXTWORD {5}
--------------------------------------------------------- overflow - c
23.PARSEOPTIONS GETNEXTWORD {5}
24.             PARSESET {19}
```

The above printout displays that the function MSPARSE calls MSINIT, MSINTERPRET, and MSPRINTSENTENCE. MSINTERPRET in turn calls MSRECORDFILE, MSPRINTWORDS, PARSECOMMAND, GETNEXTWORD, FIXUPTYPES, and FIXUPCONJUNCTIONS. The numbers in braces {} after a function name are backward references: they indicate that the tree for that function was expanded on a previous line. The lowercase letters in braces are *forward* references: they indicate that the tree for that function will be expanded below, since there is no more room on the line. The vertical bar is used to keep the output aligned.

Note: In Interlisp-D, the Browser Lispusers package modifies the SHOW PATHS command so the command's output is displayed as an undirected graph (see page 18.9).

13.2.1 Path Options

The SHOW PATHS command takes the form: SHOW PATHS followed by some combination of the following *path options:*

FROM *SET* [Masterscope Path Option]

Display the function calls from the elements of *SET*.

TO *SET* [Masterscope Path Option]

Display the function calls leading *to* elements of *SET*. If TO is given before FROM (or no FROM is given), the tree is "inverted" and a message, (inverted tree) is printed to warn the user that if FN1 appears after FN2 it is because FN1 is *called by* FN2.

When both FROM and TO are given, the first one indicates a set of functions which are to be displayed while the second restricts the paths that will be traced; i.e., the command SHOW PATHS FROM X TO Y will trace the elements of the set CALLED SOMEHOW BY X AND CALLING Y SOMEHOW.

If TO is not given, TO KNOWN OR NOT @ GETD is assumed; that is, only functions which have been analyzed or which are undefined will be included. Note that Masterscope will analyze a function while printing out the tree if that function has not previously been seen and it currently has an EXPR definition; thus, any function which *can be* analyzed will be displayed.

AVOIDING *SET* [Masterscope Path Option]

Do not display any function in *SET*. AMONG is recognized as a synonym for AVOIDING NOT. For example, SHOW PATHS TO ERROR AVOIDING ON FILE2 will not display (or trace) any function on FILE2.

NOTRACE *SET* [Masterscope Path Option]

Do not trace from any element of *SET*. NOTRACE differs from AVOIDING in that a function which is marked NOTRACE will be printed, but the tree beyond it will not be expanded; the functions in an AVOIDING set will not be printed at all. For example, SHOW PATHS FROM ANY ON FILE1 NOTRACE ON FILE2 will display the tree of calls eminating from FILE1, but will not expand any function on FILE2.

SEPARATE *SET* [Masterscope Path Option]

Give each element of *SET* a separate tree. Note that FROM and TO only insure that the designated functions will be displayed. SEPARATE can be used to guarantee that certain functions will begin new tree structures. SEPARATE functions are displayed in the same manner as overflow lines; i.e., when one of the functions indicated by SEPARATE is found, it is printed followed by a forward reference (a lower-case letter in braces) and the tree for that function is then expanded below.

LINELENGTH *N* [Masterscope Path Option]

Resets LINELENGTH to *N* before displaying the tree. The linelength is used to determine when a part of the tree should "overflow" and be expanded lower.

13.3 ERROR MESSAGES

When the user gives Masterscope a command, the command is first parsed, i.e. translated to an internal representation, and then the internal representation is interpreted. If a command cannot be parsed, e.g. if the user typed SHOW WHERE CALLED BY X, the message "Sorry, I can't parse that!" is printed and an error is generated. If the command is of the correct form but cannot be interpreted (e.g., the command EDIT WHERE ANY CONTAINS ANY) Masterscope will print the message "Sorry, that isn't implemented!" and generate an error. If the command requires that some functions having been analyzed (e.g., the command WHO CALLS X) and the database is empty, Masterscope will print the message "Sorry, no functions have been analyzed!" and generate an error.

13.4 MACRO EXPANSION

As part of analysis, Masterscope will expand the macro definition of called functions, if they are not otherwise defined (see page 5.17). Masterscope macro expansion is controlled by the variable MSMACROPROPS:

MSMACROPROPS [Variable]

Value is an ordered list of macro-property names that Masterscope will search to find a macro definition. Only the kinds of macros that appear on MSMACROPROPS will be expanded. All others will be treated as function calls and left unexpanded.

Initially (MACRO).

Note: MSMACROPROPS initially contains only MACRO (and not 10MACRO, DMACRO, etc.) in the theory that the machine-dependent macro definitions are more likely "optimizers".

Note that if you edit a macro, Masterscope will know to reanalyze the functions which call that macro. However, if your macro is of the "computed-macro" style, and it calls functions which you edit, Masterscope will not notice. You must be careful to tell masterscope to REANALYZE the appropriate functions (e.g., if you edit FOOEXPANDER which is used to expand FOO macros, you have to . REANALYZE ANY CALLING FOO.

13.5 AFFECTING MASTERSCOPE ANALYSIS

Masterscope analyzes the EXPR definitions of functions and notes in its database the relations that function has with other functions and with variables. To perform this analysis, Masterscope uses *templates* which describe the behavior of functions. For example, the information that SORT SMASHes its first argument is contained in the template for SORT. Masterscope initially contains templates for most system functions which set variables, test their arguments, or perform destructive operations.

A template is a list structure containing any of the following atoms:

PPE [in Masterscope template]

If an expression appears in this location, there is most likely a parenthesis error.

Masterscope notes this as a "call" to the function "ppe" (lowercase). Therefore, SHOW WHERE ANY CALLS ppe will print out all possible parenthesis errors. When Masterscope finds a possible parenthesis error in the course of analyzing a function definition, rather than printing the usual ".", it prints out a "?" instead.

NIL [in Masterscope template]

The expression occuring at this location is not evaluated.

SET [in Masterscope template]

A variable appearing at this place is set.

SMASH [in Masterscope template]

The value of this expression is smashed.

TEST [in Masterscope template]

This expression is used as a predicate (that is, the only use of the value of the expression is whether it is NIL or non-NIL).

PROP [in Masterscope template]

The value of this expression is used as a property name. If the expression is of the form (QUOTE *ATOM*), Masterscope will note that *ATOM* is USED AS A PROPERTY NAME. For example, the template for GETPROP is (EVAL PROP . PPE).

FUNCTION [in Masterscope template]

The expression at this point is used as a functional argument. For example, the template for MAPC is (SMASH FUNCTION FUNCTION . PPE).

FUNCTIONAL [in Masterscope template]

The expression at this point is used as a functional argument. This is like FUNCTION, except that Masterscope distinguishes between functional arguments to functions which "compile open" from those that do not. For the latter (e.g. SORT and APPLY), FUNCTIONAL should be used rather than FUNCTION.

EVAL [in Masterscope template]

The expression at this location is evaluated (but not set, smashed, tested, used as a functional argument, etc.).

RETURN [in Masterscope template]

The value of the function (of which this is the template) is the value of this expression.

TESTRETURN [in Masterscope template]

A combination of TEST and RETURN: If the value of the function is non-NIL, then it is returned. For instance, a one-element COND clause is this way.

EFFECT [in Masterscope template]

The expression at this location is evaluated, but the value is not used.

FETCH [in Masterscope template]

An atom at this location is a field which is fetched.

REPLACE [in Masterscope template]

An atom at this location is a field which is replaced.

RECORD [in Masterscope template]

An atom at this location is used as a record name.

CREATE [in Masterscope template]

An atom at this location is a record which is created.

BIND [in Masterscope template]

An atom at this location is a variable which is bound.

CALL [in Masterscope template]

An atom at this location is a function which is called.

CLISP [in Masterscope template]

An atom at this location is used as a CLISP word.

! [in Masterscope template]

This atom, which can only occur as the first element of a template, allows one to specify a template for the CAR of the function form. If ! doesn't appear, the CAR of the form is treated as if it had a CALL specified for it. In other words, the templates (.. EVAL) and (! CALL .. EVAL) are equivalent.

If the next atom after a ! is NIL, this specifies that the function name should not be remembered. For example, the template for AND is (! NIL .. TEST RETURN), which means that if you see an "AND", don't remember it as being called. This keeps the Masterscope database from being cluttered by too many uninteresting relations; Masterscope also throws away relations for COND, CAR, CDR, and a couple of others.

In addition to the above atoms which occur in templates, there are some "special forms" which are lists keyed by their CAR.

.. *TEMPLATE* [in Masterscope template]

Any part of a template may be preceded by the atom .. (two periods) which specifies that the template should be repeated an indefinite number ($N \geq 0$) of times to fill out the expression. For example, the template for COND might be (.. (TEST .. EFFECT RETURN)) while the template for SELECTQ is (EVAL .. (NIL .. EFFECT RETURN) RETURN).

(BOTH *TEMPLATE*$_1$ *TEMPLATE*$_2$) [in Masterscope template]

Analyze the current expression twice, using the each of the templates in turn.

(IF *EXPRESSION* *TEMPLATE*$_1$ *TEMPLATE*$_2$) [in Masterscope template]

Evaluate *EXPRESSION* at analysis time (the variable EXPR will be bound to the expression which corresponds to the IF), and if the result is non-NIL, use *TEMPLATE*$_1$, otherwise *TEMPLATE*$_2$. If *EXPRESSION* is a literal atom, it is APPLY'd to EXPR. For example, (IF LISTP (RECORD FETCH) FETCH) specifies that if the current expression is a list, then the first element is a record name and the second element a field name, otherwise it is a field name.

(@ *EXPRFORM* *TEMPLATEFORM*) [in Masterscope template]

Evaluate *EXPRFORM* giving *EXPR*, evaluate *TEMPLATEFORM* giving *TEMPLATE*. Then analyze *EXPR* with *TEMPLATE*. @ lets the user compute on the fly both a template and an expression to analyze it with. The forms can use the variable EXPR, which is bound to the current expression.

(MACRO . *MACRO*) [in Masterscope template]

MACRO is interpreted in the same way as a macro (see page 5.17) and the resulting form is analyzed. If the template is the atom MACRO alone, Masterscope will use the MACRO property of the function itself. This is useful when analyzing code which contains calls to user-defined macros. If the user changes a macro property (e.g. by editing it) of an atom which has template of MACRO, Masterscope will mark any function which used that macro as needing to be reanalyzed.

Some examples of templates:

function	template
DREVERSE	(SMASH . PPE)
AND	(! NIL TEST .. RETURN)
MAPCAR	(EVAL FUNCTION FUNCTION)
COND	(! NIL .. (IF CDR (TEST .. EFFECT RETURN) (TESTRETURN . PPE)))

Templates may be changed and new templates defined using the functions:

(GETTEMPLATE *FN*) [Function]

Returns the current template of *FN*.

(SETTEMPLATE *FN* *TEMPLATE*) [Function]

Changes the template for the function *FN* and returns the old value. If any functions in the database are marked as calling *FN*, they will be marked as needing re-analysis.

13.6 DATA BASE UPDATING

Masterscope is interfaced to the editor and file package so that it notes whenever a function has been changed, either through editing or loading in a new definition. Whenever a command is given which requires knowing the information about a specific function, if that function has been noted as being changed, the function is automatically re-analyzed before the command is interpreted. If the command requires that all the information in the database be consistent (e.g., the user asks WHO CALLS X) then *all* functions which have been marked as changed are re-analyzed.

13.7 MASTERSCOPE ENTRIES

(CALLS *FN* *USEDATABASE* —) [Function]

FN can be a function name, a definition, or a form. Note: CALLS will also work on compiled code. CALLS returns a list of four elements: a list of all the functions called by *FN*,[9] a list of all the variables bound in *FN*, a list of all the variables used freely in *FN*, and a list of the variables used globally in *FN*. For the purpose of CALLS, variables used freely which are on GLOBALVARS or have a property GLOBALVAR value T are considered to be used globally. If *USEDATABASE* is NIL (or *FN* is not a litatom), CALLS will perform a one-time analysis of *FN*. Otherwise (i.e. if *USEDATABASE* is non-NIL and *FN* a function name), CALLS will use the information in Masterscope's database (*FN* will be analyzed first if necessary).

(CALLSCCODE *FN* —) [Function]

The sub-function of CALLS which analyzes compiled code. CALLSCCODE returns a list of *five* elements: a list of all the functions called via "linked" function calls, a list of all functions called regularly, a list of variables bound in *FN*, a list of variables used freely, and a list of variables used globally.

(FREEVARS *FN* *USEDATABASE*) [Function]

Equivalent to (CADDR (CALLS *FN* *USEDATABASE*)). Returns the list of variables used freely within *FN*.

(MASTERSCOPE *COMMAND* —) [Function]

Top level entry to Masterscope. If *COMMAND* is NIL, will enter into a USEREXEC in which the user may enter commands. If *COMMAND* is not NIL, the command

[9] Functions called via "linked" calls from compiled code are indicated by semicolons PACKed around their name; e.g. (CALLS 'MASTERSCOPE) might return ((;MASTERSCOPEEXEC; ;MSINTERPRET; ;PRINT; HELP) --). This feature can be suppressed by setting NOPACKCALLSFLG to T.

is interpreted and MASTERSCOPE will return the value that would be printed by the command. Note that only the question commands return meaningful values.

(SETSYNONYM *PHRASE MEANING* —) [Function]

Defines a new synonym for Masterscope's parser. Both *PHRASE* and *MEANING* are lists of words; anywhere *PHRASE* is seen in a command, *MEANING* will be substituted. For example, (SETSYNONYM 'GLOBALS '(VARS IN GLOBALVARS OR @(GETPROP X 'GLOBALVAR))) would allow the user to refer with the single word GLOBALS to the set of variables which are either in GLOBALVARS or have a GLOBALVAR property.

The following functions are provided for users who wish to write their own routines using Masterscope's database:

(PARSERELATION *RELATION*) [Function]

RELATION is a relation phrase; e.g., (PARSERELATION '(USE FREELY)). PARSERELATION returns an internal representation for *RELATION*. For use in conjunction with GETRELATION.

(GETRELATION *ITEM RELATION INVERTED*) [Function]

RELATION is an internal representation as returned by PARSERELATION (if not, GETRELATION will first perform (PARSERELATION *RELATION*)); *ITEM* is an atom. GETRELATION returns the list of all atoms which have the given relation to *ITEM*. For example, (GETRELATION 'X '(USE FREELY)) returns the list of variables that X uses freely. If *INVERTED* is T, the inverse relation is used; e.g. (GETRELATION 'X '(USE FREELY) T) returns the list of functions which use X freely.

If *ITEM* is NIL, GETRELATION will return the list of atoms which have *RELATION* with *any* other item; i.e., answers the question WHO *RELATION*S ANY. Note that GETRELATION does *not* check to see if *ITEM* has been analyzed, or that other functions that have been changed have been re-analyzed.

(TESTRELATION *ITEM RELATION ITEM2 INVERTED*) [Function]

equivalent to (MEMB *ITEM2* (GETRELATION *ITEM RELATION INVERTED*)), that is, tests if *ITEM* and *ITEM2* are related via *RELATION*. If *ITEM2* is NIL, the call is equivalent to (NOT (NULL (GETRELATION *ITEM RELATION INVERTED*))), i.e., TESTRELATION tests if *ITEM* has the given *RELATION* with *any* other item.

(MAPRELATION *RELATION MAPFN*) [Function]

Calls the function *MAPFN* on every pair of items related via *RELATION*. If (NARGS *MAPFN*) is 1, then *MAPFN* is called on every item which has the given *RELATION* to *any* other item.

(MSNEEDUNSAVE *FNS MSG MARKCHANGEFLG*) [Function]

Used to mark functions which depend on a changed record declaration (or macro, etc.), and which must be LOADed or UNSAVEd (see below). *FNS* is a list of functions to be marked, and *MSG* is a string describing the records, macros, etc. on which they depend. If *MARKCHANGEFLG* is non-NIL, each function in the list is marked as needing re-analysis.

(UPDATEFN *FN EVENIFVALID* —) [Function]

Equivalent to the command ANALYZE '*FN*; that is, UPDATEFN will analyze *FN* if *FN* has not been analyzed before or if it has been changed since the time it was analyzed. If *EVENIFVALID* is set, UPDATEFN will re-analyze *FN* even if Masterscope thinks it has a valid analysis in the database.

(UPDATECHANGED) [Function]

Performs (UPDATEFN *FN*) on every function which has been marked as changed.

(MSMARKCHANGED *FN TYPE REASON*) [Function]

Mark that *FN* has been changed and needs to be reanalyzed. See MARKASCHANGED, page 11.11.

(DUMPDATABASE *FNLST*) [Function]

Dumps the current Masterscope database on the current output file in a LOADable form. If *FNLST* is not NIL, DUMPDATABASE will only dump the information for the list of functions in *FNLST*. The variable DATABASECOMS is initialized to ((E (DUMPDATABASE))); thus, the user may merely perform (MAKEFILE 'DATABASE.*EXTENSION*) to save the current Masterscope database. If a Masterscope database already exists when a DATABASE file is loaded, the database on the file will be merged with the one in core. Note that functions whose definitions are different from their definition when the database was made must be REANALYZEd if their new definitions are to be noticed.

The Databasefns package (page 23.15) provides a more convenient way of saving data bases along with the source files which they correspond to.

13.8 NOTICING CHANGES THAT REQUIRE RECOMPILING

When a record declaration, iterative statement operator or macro is changed, and Masterscope has "noticed" a use of that declaration or macro (i.e. it is used by some function known about in the data base), Masterscope will alert the user about those functions which might need to be re-compiled (e.g. they do not currently have EXPR definitions).[10] The functions which need recompiling are added to the list MSNEEDUNSAVE and a message is printed out:

```
The functions FN1, FN2,... use macros which have changed.
Call UNSAVEFNS() to load and/or unsave them.
```

In this situation, the following function is useful:

(UNSAVEFNS —) [Function]

Uses LOADFNS or UNSAVEDEF to make sure that all functions in the list MSNEEDUNSAVE have EXPR definitions, and then sets MSNEEDUNSAVE to NIL.

[10] Extra functions may be noticed; for example if FOO contains (fetch (REC X) --), and some declaration other than REC which contains X is changed, Masterscope will still think that FOO needs to be loaded/unsaved.

13.9 IMPLEMENTATION NOTES

Masterscope keeps a database of the relations noticed when functions are analyzed. The relations are intersected to form "primitive relationships" such that there is little or no overlap of any of the primitives. For example, the relation SET is stored as the union of SET LOCAL and SET FREE. The BIND relation is divided into BIND AS ARG, BIND AND NOT USE, and SET LOCAL, SMASH LOCAL, etc. Splitting the relations in this manner reduces the size of the database considerably, to the point where it is reasonable to maintain a Masterscope database for a large system of functions during a normal debugging session.

Each primitive relationship is stored in a pair of hash-tables, one for the "forward" direction and one for the "reverse". For example, there are two hash tables, USE AS PROPERTY and USED AS PROPERTY. To retrieve the information from the database, Masterscope performs unions of the hash-values. For example, to answer FOO BINDS WHO Masterscope will look in all of the tables which make up the BIND relation. The "internal representation" returned by PARSERELATION is just a list of dotted pairs of hash-tables. To perform GETRELATION requires only mapping down that list, doing GETHASH's on the appropriate hash-tables and UNIONing the result.

Hash tables are used for a variety of reasons: storage space is smaller; it is not necessary to maintain separate lists of which functions have been analyzed (a special table, DOESN'T DO ANYTHING is maintained for functions which neither call other functions nor bind or use any variables); and accessing is relatively fast. Within any of the tables, if the hash-value would be a list of one atom, then the atom itself, rather than the list, is stored as the hash-value. This also reduces the size of the database significantly.

CHAPTER 14

MISCELLANEOUS

(SYSTEMTYPE) [Function]

The SYSTEMTYPE function is intended to allow programmers to write system-dependent code. SYSTEMTYPE returns a litatom corresponding to the implementation of Interlisp: D (for Interlisp-D), TOPS-20, TENEX, JERICO, or VAX.

In Interlisp-D (and Interlisp-10), (SELECTQ (SYSTEMTYPE) ···) expressions are expanded at compile time so that this is an effective way to perform conditional compilation.

(USERNAME *A FLG*) [Function]

If *A*=NIL, returns login directory name; if *A*=T, returns connected directory name; if *A* is a number, USERNAME returns the user name corresponding to that user number.

The value is usually returned as a string. If *FLG* is a string ptr, it is smashed. If *FLG* is not a string pointer and is non-NIL, USERNAME returns the value as an atom.

(STORAGE *FLG GCFLG*) [Function]

Prints the amount of storage used for various data types. The exact printout is implementation-dependent. STORAGE returns NIL.

In Interlisp-10, the storage used by a particular type is only accurate immediately following a garbage collection of a related type. If *GCFLG*=T, STORAGE will perform the necessary garbage collections before printing its results. If *FLG*=T, includes storage used by and assigned to the system.

(DISMISS *MSECSWAIT TIMER*) [Function]

In Interlisp-10, dismisses the program for *MSECSWAIT* milliseconds, during which time the program uses no CPU time. Can be aborted by control-D, control-E, or control-B.

In Interlisp-D, dismisses the current process for *MSECSWAIT* milliseconds, using the timer *TIMER* if given (see page 14.11).

(APROPOS *STRING ALLFLG*) [Function]

(Currently only in Interlisp-D) Prints information about all litatoms in the Interlisp system which contain the string *STRING*. APROPOS will print the argument lists of litatoms with function definitions, the values of litatoms with variable bindings, and the property names defined for litatoms with property lists. If *ALLFLG* is NIL, this scan does not include "system internal" litatoms; otherwise, all litatoms are scanned.

(NEGATE *x*) [Function]

Returns the negation of *x*. For example:

```
(NEGATE '(MEMBER X Y))  =>  (NOT (MEMBER X Y))
(NEGATE '(EQ X Y))  =>  (NEQ X Y)
(NEGATE '(AND X (NLISTP X)))  =>  (OR (NULL X) (LISTP X))
```

The following two functions are useful writing programs that wish to reuse a scratch list to collect together some result (Both of these compile open):

(SCRATCHLIST *LST* x_1 x_2 $\cdots$ x_N) [NLambda NoSpread Function]

SCRATCHLIST sets up a context in which the value of *LST* is used as a "scratch" list. The expressions x_1, x_2, $\cdots$ x_N are evaluated in turn. During the course of evaluation, any value passed to ADDTOSCRATCHLIST will be saved, reusing CONS cells from the value of *LST*. If the value of *LST* is not long enough, new CONS cells will be added onto its end. If the value of *LST* is NIL, the entire value of SCRATCHLIST will be "new" (i.e. no CONS cells will be reused).

(ADDTOSCRATCHLIST *VALUE*) [Function]

For use under calls to SCRATCHLIST. *VALUE* is added on to the end of the value being collected by SCRATCHLIST. When SCRATCHLIST returns, its value is a list containing all of the things that ADDTOSCRATCHLIST has added.

14.1 SAVING INTERLISP STATE

(LOGOUT *FAST*) [Function]

Stops Interlisp, and returns control to the operating system. From there, it is possible to continue Interlisp as of the LOGOUT. LOGOUT will not affect the state of open files.

In Interlisp-D, LOGOUT writes out all altered pages from real memory to the file Lisp.virtualmem. This usually takes about 30 seconds on the Xerox 1100. If *FAST* is non-NIL, Interlisp is stopped without updating Lisp.virtualmem. Note that it will not be possible to restart Interlisp from the point of the LOGOUT, and it may not be possible to restart it at all. Typing (LOGOUT T) is preferable to just booting the machine, because it also does other cleanup operations (closing network connections, etc.).

In Interlisp-10, if Interlisp was started as a subsidiary fork (see SUBSYS, page 22.21), control is returned to the higher fork.

The function SYSOUT saves the current state of the Interlisp virtual memory on a file. The file package (page 11.1) can be used to save particular function definitions and other arbitrary objects on files, but SYSOUT saves the *total* state of the system.

The file produced by SYSOUT (known as "a sysout file", or simply "a sysout") can be restarted from the operating system (by typing LISP *SYSOUTFILE* in Interlisp-D or RUN *SYSOUTFILE* in Interlisp-10). This will restart Interlisp, and restore the virtual memory to the exact state that it had when the sysout file was made.

(SYSOUT *FILE*) [Function]

Saves the current state of the Interlisp virtual memory on the file *FILE*, in a form that can be subsequently restarted. The current state of program execution is saved in the sysout file, so (PROGN (SYSOUT 'FOO) (PRINT 'HELLO)) will cause HELLO to be printed after the sysout file is restarted.

If *FILE* is non-NIL, the variable SYSOUTFILE is set to the body of *FILE*. If *FILE* is NIL, then the value of SYSOUTFILE instead. Therefore, (SYSOUT) will save the current state on the next higher version of a file with the same name as the previous SYSOUT. Also, if the extension for *FILE* is not specified, the value of SYSOUT.EXT is used. This is initially SYSOUT in Interlisp-D, SAV in Tenex Interlisp-10, and EXE in Tops-20 Interlisp-10.

SYSOUT sets SYSOUTDATE to (DATE), the time and date that the SYSOUT was performed.

If SYSOUT was not able to create the sysout file, because of disk or computer error, or because there was not enough space on the directory, SYSOUT returns NIL. Otherwise it returns the full file name of *FILE*.

Actually, SYSOUT "returns" twice; when the sysout file is first created, and when it is subsequently restarted. In the latter case, SYSOUT returns the list (*FILE* . *MAKESYSFILE*), where *FILE* is the sysout file, and *MAKESYSFILE* is the original Interlisp makesys file (see MAKESYS, below). For example, (if (LISTP (SYSOUT 'FOO)) then (PRINT 'HELLO)) will cause HELLO to be printed when the sysout file is restarted, but not when SYSOUT is initially performed.

Note: SYSOUT does not save the state of any open files. WHENCLOSE (page 6.11) can be used to associate certain operations with open files so that when a SYSOUT is started up, these files will be reopened, and file pointers repositioned.

In Interlisp-10, a sysout file only contains the parts of the virtual memory that the user has changed. When the sysout file is restarted, the other pages are taken from the makesys file of the Interlisp system within which the sysout file was made (see MAKESYS, below). Therefore, whenever the Interlisp system is reassembled and/or reloaded, old sysout files are *not* compatible with the new system.

In Interlisp-D, a sysout file contains a copy of the entire allocated virtual memory, so it is very large. A normal sized sysout file contains about 4000 pages. Unlike in Interlisp-10, a sysout file is copied into the virtual memory when it is restarted, to it is perfectly permissible to overwrite a sysout file on top of the currently running sysout, for example, (SYSOUT '{DSK}FOO.SYSOUT;1) to overwrite FOO.SYSOUT on the local disk. Not only is this permissible, it is much faster than making a new sysout file (almost twice as fast, due to less disk overhead). Making a sysout file on the Xerox 1100 currently takes at least 5 minutes.

SYSOUT evaluates the expressions on BEFORESYSOUTFORMS before creating the sysout file. This variable initially includes expressions to: (1) Set the variables SYSOUTDATE and SYSOUTFILE as described above; (2) Default the sysout file name *FILE* according to the values of the variables SYSOUTFILE and SYSOUT.EXT, as described above; and (3) Perform any necessary operations on open files as specified by calls to WHENCLOSE (page 6.11).

After a sysout file is restarted (but *not* when it is initially created), SYSOUT evaluates the expressions on AFTERSYSOUTFORMS. This initially includes expressions to: (1) Perform any necessary operations on previously-opened files as specified by calls to WHENCLOSE (page 6.11); (2) [Interlisp-10 only] Reset the terminal line length with SETLINELENGTH (page 6.8); (3) [Interlisp-10 only] Reset the terminal control characters using SETTERMCHARS (page 17.59) if the operating system has changed from Tenex to Tops-20 or vice versa; (4) Possibly print a message, as determined by the value of SYSOUTGAG (see below); and (5) Call SETINITIALS to reset the initials used for time-stamping (page 17.60).

SYSOUTGAG [Variable]

The value of SYSOUTGAG determines what is printed when a sysout file is restarted. If the value of SYSOUTGAG is a list, the list is evaluated, and no additional message is printed. This allows the user to print a message. If SYSOUTGAG is non-NIL and not a list, no message is printed. Finally, if SYSOUTGAG is NIL (its initial value), and the sysout file is being restarted by the same user that made the sysout originally, the user is greeted by printing the value of HERALDSTRING (see below) followed by a greeting message. If the SYSOUT file was made by a different user, a message is printed, warning that the user profiles may be different (see page 14.5);

(SYSIN *FILE*) [Function]

[Interlisp-10 only] Restores the state of Interlisp from a sysout file. This is essentially the same as exiting Interlisp, and restarting a sysout file from the operating system executive. If SYSIN returns NIL, there was a problem in reading the file. If *FILE* was not found, generates a FILE NOT FOUND error.

(SYSOUTP *FILE*) [Function]

[Interlisp-10 only] Returns the name of the original Interlisp makesys file (see MAKESYS, below) if *FILE* is a sysout file, otherwise NIL.

FILE may also be a JFN.

(MAKESYS *FILE* *NAME*) [Function]

Used to store a new Interlisp system on the "makesys file" *FILE*. Before this is done, the system is "initialized" by undoing the greet history, and clearing the display [Interlisp-D].

When the system is first started up, a "herald" is printed identifying the system, typically "Interlisp-*xx* *DATE* ...". If *NAME* is non-NIL, MAKESYS will use it instead of Interlisp-*xx* in the herald. MAKESY.S sets HERALDSTRING to the herald string printed out.

MAKESYS also sets the variable MAKESYSDATE to (DATE), i.e. the time and date the system was made.

In Interlisp-D, MAKESYS is almost the same as SYSOUT, except that it does some cleaning-up operations (such as clearing the screen). In Interlisp-10, however, MAKESYS is considerably different from SYSOUT, because it saves *all* of the pages in the Interlisp virtual memory, and allows the makesys file to be shared between multiple users.

The Interlisp-10 system initially obtained by the user is shared; that is, all active users of Interlisp-10 are actually using the same pages of memory. As a user adds to the system, private pages are added to his memory. Similarly, if the user changes anything in the original shared Interlisp-10, for example, by advising a system function, a private copy of the changed page is created.

In addition to the swapping time saved by having several users accessing the same memory, the sharing mechanism permits a large saving in garbage collection time, since it is not necessary to garbage collect any data in the shared system, and thus Interlisp-10 does not need to chase from any pointers on shared pages during garbage collections.

This reduction in garbage collection time is possible because the shared system usually is not modified very much by the user. If the shared system is changed extensively, the savings in time will vanish, because once a page that was initially shared is made private, every pointer on it must be assumed active, because it may be pointed *to* by something in the shared system. Since every pointer on an initially shared but now private page can also point to *private* data, they must always be chased.

A user may create his own shared system with the function MAKESYS. If several people are using the same system, making the system be shared will result in a savings in swapping time. Similarly, if a system is large and seldom modified, making it be shared will result in a reduction of garbage collection time, and may therefore be worthwhile even if the system is only being used by one user.

One problem with using MAKESYS in Interlisp-10 is that it may protect large amounts of useless data from being garbage collected. For example, suppose that during the course of building an Interlisp system, a large number of list cells are used and discarded. If MAKESYS is now executed to store the system, all of that list cell space is stored, and protected from garbage collection (unless the user changes those pages, making a personal copy). To solve this problem, it is necessary to make sure that as little storage as possible is allocated while creating a new system, perhaps by setting MINFS (page 22.10) to a very low value. Of course, this will slow down Interlisp considerably, so making a new system will take a long time.

14.2 GREETING AND USER PROFILES

Many of the features of Interlisp are parameterized to allow the user to adjust the system to his or her own tastes. Among the more commonly adjusted parameters are PROMPT#FLG (page 8.18), DWIMWAIT (page 15.11), CHANGESLICE (page 8.18), LOWERCASE (page 16.21), #UNDOSAVES (page 8.33), INITIALSLST (page 17.60), etc. In addition, the user can modify the action of system functions in ways not specifically provided for by using ADVISE (page 10.9).

In order to encourage this procedure, and to make it as painless and automatic as possible, the programmeer's assistant includes a facility for both a site-defined profile and a user-defined profile. When Interlisp is first run, it calls the function GREET (see below). This provides a way of setting defaults for a particular community of users, patching bugs, etc.

Greeting (i.e., the initialization) is undoable, and is stored as a separate event on the history list (page 8.25). The user can explicitly invoke the greeting operation at any time via the function GREET. This can also be use to effect another user's initialization.

(GREET *NAME* —) [Function]

Performs the greeting for the user whose username is *NAME* (if *NAME*=NIL, uses the login name). When Interlisp first starts up, it performs (GREET).

Before GREET performs the indicated initialization, it first undoes the effects of the previous greeting. The side effects of the greeting operation are stored on a global variable as well as the history list, thus enabling the previous greeting to be undone even if it is no longer on the history list. In addition, MAKESYS is advised to undo the effects of the previous greeting, thereby returning the system to a pristine state.

GREET initializes in the following way: It first evaluates each item in the list PREGREETFORMS, then it loads the file returned from (GREETFILENAME T), then it loads the file returned from (GREETFILENAME *USERNAME*), then it evaluates each item on POSTGREETFORMS, and finally it prints a greeting such as "Hello, *xxx*.", where *xxx* is the *FIRSTNAME* component of the user's entry on INITIALSLST (page 17.60). The loads are performed "silently" by rebinding PRETTYHEADER (page 11.36) to NIL.

(GREETFILENAME *USER*) [Function]

GREETFILENAME is a system-dependent function. Its purpose is to locate existing files used for greeting and return them. If *USER* is T, then it returns the filename of the site-defined profile (if it exists). Otherwise, *USER* is interpreted to be a user's system name, and it returns the filename for the user-defined profile (if it exists).

GREETDATES [Variable]

The value of GREETDATES can be used to specify special greeting messages for various dates. GREETDATES is a list of elements of the form (*DATESTRING* . *STRING*), e.g. ("25-DEC" . "Merry Christmas"). The user can add entries to this list in his/her INIT.LISP file by using a ADDVARS file package command like (ADDVARS (GREETDATES ("8-FEB" . "Happy Birthday"))). On the specified date, the GREET will use the indicated salutation.

14.3 MANIPULATING FILE DIRECTORIES

The following function allows the user to conveniently specify and/or program a variety of directory operations:

(DIRECTORY *FILES COMMANDS DEFAULTEXT DEFAULTVERS*) [Function]

FILES is either [1] NIL (which is equivalent to *.*;*); or [2] an atom which can contain $'s or *'s (equivalent) which match any number of characters or ?'s which match a single character, or else [3] *FILES* is a list of the form (*FILES* + *FILES*), (*FILES* - *FILES*), or (*FILES* * *FILES*),[1] e.g., (T$ + $L) will match with any file beginning with T or ending in L, (T$ - *.DCOM) matches all files that begin with T and are not .DCOM files.

For each file that matches, each command in *COMMANDS* is executed with the following interpretation:

P Print file name.

PP Print file name (except for version number).

a string Prints the string.

[1] OR can be used for +, and AND for *.

READDATE, WRITEDATE, CREATIONDATE
SIZE, LENGTH, BYTESIZE
PROTECTION, AUTHOR, TYPE
Prints the appropriate information returned by GETFILEINFO (page 6.6).

COLLECT The value of DIRECTORY will be a list of file names; add the complete file name of this file to that list.

COUNTSIZE The value of DIRECTORY will be a sum; add the size of this file to that sum.

PAUSE Wait until the user types any char before proceeding with the rest of the commands (good for display if you want to ponder).

PROMPT *MESS* Prompts with *MESS*; if user responds with No, abort command processing for this file.

OLDERTHAN *N* Continue command processing if the file hasn't been referenced (read or written) in *N* days.

OLDVERSIONS *N* Continue command processing if there are at least *N* more recent versions of the same file.

BY *USER* Continue command processing if the file was last written by the given user.

@ *X* *X* is either a function of one argument (*JFN*), a function of two arguments (*JFN FILENAME*) or an arbitrary expression which uses the variablers JFN and/or the variables FILENAME freely. If *X* returns NIL, abort command processing for this file.

DELETED Allows DIRECTORY to examine deleted files (normally, they are not mapped over.

OUT *FILE* Directs output to *FILE*.

COLUMNS *N* Attempt to format output in *N* columns (rather than just 1).

TRIMTO *N* Deletes all but *N* versions of file ($N \geq 0$).

DELETE Deletes file. If this is specified, the value of DIRECTORY is NIL if no COLLECT command is specified, otherwise the list of files deleted.

UNDELETE Undeletes the indicated files that have been deleted.

DIRECTORY uses DIRCOMMANDS to correct spelling, which also provides a way of defining abbreviations and synonyms (page 15.13). Currently the following abbreviations are recognized:

AU	=>	AUTHOR
-	=>	PAUSE
COLLECT?	=>	PROMPT " ? " COLLECT
DA TI	=>	WRITEDATE

```
DEL         =>    DELETE

DEL?
DELETE?     =>    PROMPT " delete? " DELETE

OLD         =>    OLDERTHAN 90

PR          =>    PROTECTION

SI          =>    SIZE
```

(FILDIR *FILEGROUP* —) [Function]

FILEGROUP is a file group descriptor, i.e., it can contain stars. FILDIR returns a list of the files which match *FILEGROUP*, a la the DIRECTORY function, e.g., (FILDIR '*.COM;0).

There is also a programmer's assistant command DIR which calls the function DIRECTORY:

DIR *FILES* . *COMMANDS* [Prog. Asst. Command]

Calls the function DIRECTORY with (P . *COMMANDS*) as the command list and * and * as the default extension and default version respectively.

For example, to DELVER only those files which you ok, do DIR *FILES* PROMPT "?" TRIMTO 1.

14.4 SORTING LISTS

(SORT *DATA* *COMPAREFN*) [Function]

DATA is a list of items to be sorted using *COMPAREFN*, a predicate function of two arguments which can compare any two items on *DATA* and return T if the first one belongs before the second. If *COMPAREFN* is NIL, ALPHORDER is used; thus (SORT *DATA*) will alphabetize a list. If *COMPAREFN* is T, CAR's of items that are lists are given to ALPHORDER, otherwise the items themselves; thus (SORT A-LIST T) will alphabetize an assoc list by the CAR of each item. (SORT X 'ILESSP) will sort a list of integers.

The value of SORT is the sorted list. The sort is destructive and uses no extra storage. The value returned is EQ to *DATA* but elements have been switched around. Interrupting with control D, E, or B may cause loss of data, but control H may be used at any time, and SORT will break at a clean state from which ↑ or control characters are safe. The algorithm used by SORT is such that the maximum number of compares is $N*\log_2 N$, where N is (LENGTH *DATA*).

Note: if (*COMPAREFN* A B) = (*COMPAREFN* B A), then the ordering of A and B may or may not be preserved.

For example, if (FOO . FIE) appears before (FOO . FUM) in X, (SORT X T) may or may not reverse the order of these two elements. Of course, the user can always specify a more precise *COMPAREFN*.

(MERGE *A B COMPAREFN*) [Function]

A and *B* are lists which have previously been sorted using SORT and *COMPAREFN*. Value is a destructive merging of the two lists. It does not matter which list is longer. After merging both *A* and *B* are equal to the merged list. (In fact, (CDR *A*) is EQ to (CDR *B*)). MERGE may be aborted after control-H.

(ALPHORDER *A B*) [Function]

A predicate function of two arguments, for alphabetizing. Returns T if its arguments are in order, i.e., if *B* does not belong before *A*. Numbers come before literal atoms, and are ordered by magnitude (using GREATERP). Literal atoms and strings are ordered by comparing the character codes in their pnames. Thus (ALPHORDER 23 123) is T, whereas (ALPHORDER 'A23 'A123) is NIL, because the character code for the digit 2 is greater than the code for 1.

Atoms and strings are ordered before all other data types. If neither *A* nor *B* are atoms or strings, the value of ALPHORDER is T, i.e., in order.

Note: ALPHORDER does no UNPACKs, CHCONs, CONSes or NTHCHARs. It is several times faster for alphabetizing than anything that can be written using these other functions.

(MERGEINSERT *NEW LST ONEFLG*) [Function]

LST is NIL or a list of partially sorted items. MERGEINSERT tries to find the "best" place to (destructively) insert *NEW*, e.g.,

```
(MERGEINSERT 'FIE2 '(FOO FOO1 FIE FUM))
       =>   (FOO FOO1 FIE FIE2 FUM)
```

Returns *LST*. MERGEINSERT is undoable.

If *ONEFLG*=T and *NEW* is already a member of *LST*, MERGEINSERT does nothing and returns *LST*.

MERGEINSERT is used by ADDTOFILE (page 11.33) to insert the name of a new function into a list of functions. The algorithm is essentially to look for the item with the longest common leading sequence of characters with respect to *NEW*, and then merge *NEW* in starting at that point.

(COMPARELISTS *X Y*) [Function]

Compares *X* and *Y* and prints their differences, i.e., COMPARELISTS is essentially a SRCCOM for list structures.

14.5 DATE/TIME FUNCTIONS

(DATE —) [Function]

Obtains date and time, returning it as a single string with format "*DD*-*MM*-*YY* *HH*:*MMM*:*SS*", where *DD* is day, *MM* is month, *YY* year, *HH* hours, *MMM* minutes, *SS* seconds, e.g., "14-MAY-71 14:26:08".

In Interlisp-10, DATE will accept *FORMATBITS* as an argument, which can be used to specify other formats, e.g., day of week, time zone, etc., as described in the JSYS manual.

(IDATE *STR*) [Function]

STR is a date and time string. Value of IDATE is *STR* converted to a number such that if *DATE*$_1$ is before (earlier than) *DATE*$_2$, then (IDATE *DATE*$_1$) < (IDATE *DATE*$_2$). (IDATE) returns (IDATE (DATE)).

(GDATE *DATE FORMATBITS STRPTR*) [Function]

Interlisp-10 function for obtaining time-date formatted string, *DATE* is in internal date-and-time format. If NIL, current time and date is used, i.e. value of (IDATE). *FORMATBITS* is 36 bit quantity to be passed to TENEX/TOPS 20 time-date conversion routines (see JSYS manual). For example, *FORMATBITS*=-1 gives a "long" date, e.g. "FRIDAY, JUN 16, 1978, 23:41:52-PDT". If *FORMATBITS*=NIL, defaults to a value which will produce the same format as that of (DATE), i.e. "*DD-MM-YY HH:MMM:SS*". *STRPTR* is an optional string pointer to be reused. In this case, the string characters are stored in an internal scratch string, MACSCRATCHSTRING, so that a subsequent call to GDATE will overwrite the characters returned by this one. Note that this internal scratch string is also used by several other functions in this section.

The dateformat package (page 23.57) provides a convenient way of specifying the format bits in terms of keywords.

(CLOCK *N* —) [Function]

For *N*=0, returns the current value of the time of day clock i.e., number of milliseconds since last system start up.

For *N*=1, returns the value of the time of day clock when the user started up this Interlisp, i.e., difference between (CLOCK 0) and (CLOCK 1) is number of milliseconds (real time) since this Interlisp was started.

For *N*=2, returns the number of milliseconds of *compute* time since user started up this Interlisp (garbage collection time is subtracted off).

For *N*=3, returns the number of milliseconds of compute time spent in garbage collections (all types).[2]

14.6 TIMERS AND DURATION FUNCTIONS

Often one needs to loop over some code, stopping when a certain interval of time has passed. Some systems provide an "alarmclock" facility, which provides an asynchronous interrupt when a time interval runs out. This is not particularly feasible in the current Interlisp-D envirornment, so the following facilities are supplied for efficiently testing for the expiration of a time interval in a loop context.

[2] In Interlisp-10, this number is directly accessible via the COREVAL GCTIM.

Three functions are provided: SETUPTIMER, SETUPTIMER.DATE, and TIMEREXPIRED?. Also several new i.s.oprs have been defined: forDuration, during, untilDate, timerUnits, usingTimer, and resourceName (reasonable variations on upper/lower case are permissible).

These functions use an object called a Timer, which encodes a future clock time at which a signal is desired. A Timer is constructed by the functions SETUPTIMER and SETUPTIMER.DATE, and is created with a basic clock "unit" selected from among SECONDS, MILLISECONDS, or TICKS. The first two timer units provide a machine/system independent interface, and the latter provides access to the "real", basic strobe unit of the machine's clock on which the program is running. The default unit is MILLISECONDS.

Currently, the TICKS unit is the same as the MILLISECONDS unit for Interlisp-10 and Interlisp/VAX. In Interlisp-D, the TICKS unit is a function of the particular machine that Interlisp-D is running on: The Xerox 1100 and 1132 have about 0.5952 microseconds per tick (1680 ticks per millisecond); The Xerox 1108 has about 28.78 microseconds per tick (34.746 ticks per millisecond). The advantage of using TICKS rather than one of the uniform interfaces is primarily speed; e.g., on a Xerox 1100, it may take as much as 400 microseconds to interface the milliseconds clock (a software facility actually based over the real clock), whereas reading the real clock itself should take less than about ten microseconds. The disadvantage of the TICKS unit is its short roll-over interval (about 20 minutes) compared to the MILLISECONDS roll-over interval (about about two weeks), and also the dependency on particular machine parameters.

(SETUPTIMER *INTERVAL OLDTIMER? TIMERUNITS INTERVALUNITS*) [Function]

SETUPTIMER returns a Timer that will "go off" (as tested by TIMEREXPIRED?) after a specified time-interval measured from the current clock time. SETUPTIMER has one required and three optional arguments:

INTERVAL must be a integer specifying how long an interval is desired. *TIMERUNITS* specifies the units of measure for the interval (defaults to MILLISECONDS).

If *OLDTIMER?* is a Timer, it will be reused and returned, rather than allocating a new Timer. *INTERVALUNITS* specifies the units in which the *OLDTIMER?* is expressed (defaults to the value of *TIMERUNITS*.

(SETUPTIMER.DATE *DTS OLDTIMER?*) [Function]

SETUPTIMER.DATE returns a Timer (using the SECONDS time unit) that will "go off" at a specified date and time. *DTS* is a Date/Time string such as IDATE accepts (page 14.10). If *OLDTIMER?* is a Timer, it will be reused and returned, rather than allocating a new Timer.

SETUPTIMER.DATE operates by first subtracting (IDATE) from (IDATE *DTS*), so there may be some large integer creation involved, even if *OLDTIMER?* is given.

(TIMEREXPIRED? *TIMER CLOCKVALUE.OR.TIMERUNITS*) [Function]

If *TIMER* is a Timer, and *CLOCKVALUE.OR.TIMERUNITS* is the time-unit of *TIMER*, TIMEREXPIRED? returns true if *TIMER* has "gone off".

CLOCKVALUE.OR.TIMERUNITS can also be a Timer, in which case TIMEREXPIRED? compares the two timers (using the same time units). If X and Y are Timers, then (TIMEREXPIRED? X Y) is true if X is set for a *later* time than Y.

There are a number of i.s.oprs that make it easier to use Timers in iterative statements (page 4.5). These i.s.oprs are given below in the "canonical" form, with the second "word" capitalized, but the all-caps and all-lower-case versions are also acceptable.

`forDuration` *INTERVAL* [I.S. Operator]

`during` *INTERVAL* [I.S. Operator]

INTERVAL is an integer specifying an interval of time during which the iterative statement will loop.

`timerUnits` *UNITS* [I.S. Operator]

UNITS specifies the time units of the *INTERVAL* specified in `forDuration`.

`untilDate` *DTS* [I.S. Operator]

DTS is a Date/Time string (such as `IDATE` accepts) specifying when the iterative statement should stop looping.

`usingTimer` *TIMER* [I.S. Operator]

If `usingTimer` is given, *TIMER* is reused as the timer for `forDuration` or `untilDate`, rather than creating a new timer. This can reduce allocation if one of these i.s.oprs is used within another loop.

`resourceName` *RESOURCE* [I.S. Operator]

RESOURCE specifies a `GLOBALRESOURCES` name to be used as the timer storage. If *RESOURCE*=`T`, it will be converted to a common internal name.

Some examples:

```
(during 6MONTHS  timerUnits 'SECS
 until (TENANT-VACATED? HouseHolder)
 do (DISMISS <for-about-a-day>)
    (HARRASS HouseHolder)
 finally (if (NOT (TENANT-VACATED? HouseHolder))
          then (EVICT-TENANT HouseHolder)))
```

This humorous little example shows that how is is possible to have two termination condition: (1) when the time interval of `6MONTHS` has elapsed, or (2) when the predicate `(TENANT-VACATED? HouseHolder)` becomes true. Note that the "finally" clause is executed regardless of which termination condition caused it.

```
(do (forDuration (CONSTANT (ITIMES 10 24 60 60 1000))
     do (CARRY.ON.AS.USUAL)
     finally (PROMPTPRINT "Have you had your 10-day check-up?")))
```

This infinite loop breaks out with a warning message every 10 days. One could question whether the millisecond clock, which is used by default, is appropriate for this loop, since it rolls-over about every two weeks.

```
(SETQ \RandomTimer (SETUPTIMER 0))
(untilDate "31-DEC-83 23:59:59"  usingTimer \RandomTimer
 when (WINNING?) do (RETURN)
 finally (ERROR "You've been losing this whole year!"))
```

Here we see a usage of an explicit date for the time interval; also, the user has squirreled away some storage (as the value of `\RandomTimer`) for use by the call to `SETUPTIMER` in this loop.

```
(forDuration SOMEINTERVAL
 resourcename '\INNERLOOPBOX
 timerunits 'TICKS
 do (CRITICAL.INNER.LOOP))
```

For this loop, the user doesn't want any CONSing to take place, so \INNERLOOPBOX will be defined as a GLOBALRESOURCES which "caches" a timer cell (if it isn't already so defined), and wraps the entire statement in a GLOBALRESOURCE call. Furthermore, he has specified a time unit of TICKS, for lower overhead in this critical inner loop. In fact specifying a resourcename of T would have been the same as specifying it to be \ForDurationOfBox; this is just a simpler way to specify that a GLOBALRESOURCE is wanted, without having to think up a name.

14.7 GAINSPACE

For users with large programs and data bases, the user may sometimes find himself in a situation where he needs to obtain more space, and is willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The following function is available for this purpose.

(GAINSPACE) [Function]

Prints a list of deletable objects, allowing the user to specify at each point what should be discarded and what should be retained.

For example:

```
← (GAINSPACE)
purge history lists ? Yes
purge everything, or just the properties, e.g., SIDE, LISPXPRINT, etc. ?
just the properties
discard definitions on property lists ? Yes
discard old values of variables ? Yes
erase properties ? No
erase CLISP translations? Yes
   .
   .
   .
```

GAINSPACE is driven by the list GAINSPACEFORMS. Each element on GAINSPACEFORMS is of the form (*PRECHECK MESSAGE FORM KEYLST*). If *PRECHECK*, when evaluated, returns NIL, GAINSPACE skips to the next entry. For example, the user will not be asked whether or not to purge the history list if it is not enabled. Otherwise, ASKUSER (page 6.57) is called with the indicated *MESSAGE* and the (optional) *KEYLST*. If the user responds No, i.e., ASKUSER returns N, GAINSPACE skips to the next entry. Otherwise, *FORM* is evaluated with the variable RESPONSE bound to the value of ASKUSER. In the above example, the *FORM* for the "purge history lists" question calls ASKUSER to ask "purge everything, ···" only if the user had responded Yes. If the user had responded with Everything, the second question would not have been asked.

The "erase properties" question is driven by a list SMASHPROPSMENU. Each element on this list is of the form (*MESSAGE* . *PROPS*). The user is prompted with *MESSAGE* (by ASKUSER), and if he responds Yes, *PROPS* is added to the list SMASHPROPS. The "discard definitions on property lists" and "discard old values of variables" questions also add to SMASHPROPS. The user will not be prompted for any entry on SMASHPROPSMENU for which all of the corresponding properties are already on SMASHPROPS. SMASHPROPS is initially set to the value of SMASHPROPSLST. This permits the user to specify in advance those properties which he always wants to be discarded, and not be asked about them subsequently. After finishing all the entries on GAINSPACEFORMS, GAINSPACE checks to see if the value of SMASHPROPS is non-NIL, and if so, does a MAPATOMS, i.e., looks at every atom in the system, and erases the indicated properties.

Note that the user can change or add new entries to GAINSPACEFORMS or SMASHPROPSMENU, so that GAINSPACE can also be used to purge structures that the user's programs have accumulated.

14.8 PERFORMANCE MEASURING FUNCTIONS

(CONSCOUNT *N*) [Function]

(CONSCOUNT) returns the number of CONSes since Interlisp started up. If *N* is not NIL, resets CONSCOUNT to *N*.

(BOXCOUNT *TYPE N*) [Function]

Returns the number of boxing operations for the data type *TYPE* (see page 2.36) since Interlisp started up. If *N* is not NIL, the corresponding counter is reset to *N*.

In Interlisp-10, if *TYPE*=NIL, BOXCOUNT returns the number of large integer boxes; if *TYPE* is non-NIL, it returns the number of floating boxes. These counters are directly accessible via the COREVALs IBOXCN and FBOXCN.

In Interlisp-D, *TYPE* can be any datatype name, in addition to FIXP and FLOATP.

(PAGEFAULTS) [Function]

Returns the number of page faults since Interlisp started up.

(TIME *TIMEX TIMEN TIMETYPE*) [NLambda Function]

An nlambda function. It executes the computation *TIMEX*, and prints out the number of conses and computation time. Garbage collection time is subtracted out. For example, in Interlisp-10:

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP]
FILE CREATED 1-AUG-78 14:56:12
PRETTYCOMS
collecting lists
582, 10291 free cells
13169 CONSES
29.484 SECONDS
PRETTY
```

If *TIMEN* is greater than 1 (*TIMEN*=NIL is equivalent to *TIMEN*=1), *TIMEX* is executed *TIMEN* times, and TIME prints out (number of conses)/*TIMEN*, and (computation time)/*TIMEN*. This is useful for more accurate measurement on small computations, e.g.

```
←TIME((COPY (QUOTE (A B C))) 10)
30/10 = 3 CONSES
.055/10 = .0055 SECONDS
(A B C)
```

If *TIMETYPE* is 0, TIME measures and prints total *real* time as well as computation time, e.g.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 0]
FILE CREATED 7-MAY-71 12:47:14
GC: 8
582, 10291 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
11.193 SECONDS
27.378 SECONDS, REAL TIME
PRETTY
```

If *TIMETYPE* = 3, TIME measures and prints garbage collection time as well as computation time, e.g.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 3]
FILE CREATED 7-MAY-71 12:47:14
GC: 8
582, 1091 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
10.597 SECONDS
1.487 SECONDS, GARBAGE COLLECTION TIME
PRETTY
```

Another option is *TIMETYPE*=T, in which case TIME measures and prints the number of pagefaults.

The value of TIME is the value of the last evaluation of *TIMEX*.

14.8.1 BREAKDOWN

TIME collects statistics for whole computations. BREAKDOWN is available to analyze the breakdown of computation time (or any other measureable quantity) function by function.

(BREAKDOWN *FN*$_1$ ··· *FN*$_N$) [NLambda NoSpread Function]

The user calls BREAKDOWN giving it a list of function names (unevaluated). These functions are modified so that they keep track of various statistics.

To remove functions from those being monitored, simply UNBREAK (page 10.6) the functions, thereby restoring them to their original state. To add functions, call BREAKDOWN on the new functions. This will not reset the counters for any functions not on the new list. However (BREAKDOWN) will zero the counters of all functions being monitored.

The procedure used for measuring is such that if one function calls other and both are "broken down", then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

Note: BREAKDOWN will *not* give accurate results if a function being measured is not returned from normally, e.g., a lower RETFROM (or ERROR) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

(BRKDWNRESULTS *RETURNVALUESFLG*) [Function]

BRKDWNRESULTS prints the analysis of the statistics requested as well as the number of calls to each function. If *RETURNVALUESFLG* is non-NIL, BRKDWNRESULTS will not to print the results, but instead return them in the form of a list of elements of the form (*FNNAME #CALLS VALUE*).

Example:

```
← (BREAKDOWN SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
← (PRETTYDEF '(SUPERPRINT) 'FOO)
FOO.;3
← (BRKDWNRESULTS)
FUNCTIONS      TIME       #CALLS    PER CALL     %
SUPERPRINT     8.261       365       0.023      20
SUBPRINT       31.910      141       0.226      76
COMMENT1       1.612       8         0.201      4
TOTAL          41.783      514       0.081
NIL
← (BRKDWNRESULTS T)
((SUPERPRINT 365 8261) (SUBPRINT 141 31910) (COMMENT1 8 1612))
```

BREAKDOWN can be used to measure other statistics, by setting the following variables:

BRKDWNTYPE [Variable]

To use BREAKDOWN to measure other statistics, before calling BREAKDOWN, set the variable BRKDWNTYPE to the quantity of interest, e.g., TIME, CONSES, etc, or a list of such quantities. Whenever BREAKDOWN is called with BRKDWNTYPE not NIL, BREAKDOWN performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it. The functions being broken down will be redefined to call this measuring function. When BREAKDOWN is through initializing, it sets BRKDWNTYPE back to NIL. Subsequent calls to BREAKDOWN will measure the new statistic until BRKDWNTYPE is again set and a new BREAKDOWN performed.

BRKDWNTYPES [Variable]

The list BRKDWNTYPES contains the information used to analyze new statistics. Each entry on BRKDWNTYPES should be of the form (*TYPE FORM FUNCTION*), where *TYPE* is a statistic name (as would appear in BRKDWNTYPE), *FORM* computes the statistic, and *FUNCTION* (optional) converts the value of form to some more interesting quantity. For example, (TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000))) measures computation time and reports the result in seconds instead of milliseconds. BRKDWNTYPES currently contains entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

Example:

```
← (SETQ BRKDWNTYPE '(TIME CONSES))
(TIME CONSES)
← (BREAKDOWN MATCH CONSTRUCT)
(MATCH CONSTRUCT)
← (FLIP '(A B C D E F G H C Z) '(.. $1 .. #2 ..) '(.. #3 ..))
(A B D E F G H Z)
← (BRKDWNRESULTS)
FUNCTIONS     TIME        #CALLS    PER CALL    %
MATCH         0.036        1         0.036     54
CONSTRUCT     0.031        1         0.031     46
TOTAL         0.067        2         0.033
FUNCTIONS     CONSES     #CALLS    PER CALL     %
MATCH         32           1        32.000     40
CONSTRUCT     49           1        49.000     60
TOTAL         81           2        40.500
NIL
```

Occasionally, a function being analyzed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If the user were using TIME, he would specify a value for *TIMEN* greater than 1 to give greater accuracy. A similar option is available for BREAKDOWN. The user can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to BREAKDOWN, e.g., BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for EDITCOM and EDIT4F but executes (the body of) EDIT4E and EQP 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from BRKDWNRESULTS will look the same as though each function were run only once, except that the measurement will be more accurate.

Another way of obtaining more accurate measurement is to expand the call to the measuring function in-line. If the value of BRKDWNCOMPFLG is non-NIL (initially NIL), then whenever a function is broken-down, it will be redefined to call the measuring function, and then recompiled. The measuring function is expanded in-line via an appropriate macro. In addition, whenever BRKDWNTYPE is reset, the compiler is called for *all* functions for which BRKDWNCOMPFLG was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

14.9 PAGE MAPPED FILES

This facility allows paged access to files. It manages a set of paging buffers as a least-recently-used queue, with each buffer being a full-page block. Facilities are provided for allocating and deallocating buffers, locking down pages, mapping a given page of the file into core, and getting the in-core location to which a given word of the file has been mapped. Any number of files can be mapped in at one time.

Note: Interlisp-D implements the page-mapping primitives of Interlisp-10 with some notable differences that might require major reworking of programs that rely on these facilities. The major difference is that an Interlisp-D page contains 256 16-bit words, rather than the 512 36-bit words of Interlisp-10. A given page number or file address for MAPPAGE or MAPWORD will correspond to a very different number of bits from the beginning of the file, and WORDCONTENTS and SETWORDCONTENTS move smaller amounts of information. A second difference is that buffers are completely integrated into the Interlisp-D storage management system so that a page is guaranteed to be locked down as long as the user holds a pointer to it. The functions LOCKMAP and UNLOCKMAP are therefore unnecessary, but for compatibility are defined with dummy definitions.

The following scenario illustrates the use of these facilities: The user first opens the file (or files) that he wants to access by page-mapping using any of the ordinary file-opening functions. Then, to examine a particular word in one of the files, the user simply gives the word number and the file's name to the function MAPWORD, which returns a pointer to the in-core location that that word is mapped to (i.e. the address as an unboxed number). When he has finished processing, the user simply closes the file (e.g. using CLOSEF) and the buffers are automatically unmapped.

The basic functions are:

(ADDMAPBUFFER *TEMP ERRORFLG*) [Function]

Initially, a single buffer is allocated, so that page-mapping may be done without further initialization. More buffers can be allocated by ADDMAPBUFFER, which may help to avoid thrashing. ADDMAPBUFFER attempts to allocate a single new buffer, and returns non-NIL if successful. If there is not enough space to allocate a new buffer, then if *ERRORFLG* is NIL, ADDMAPBUFFER simply returns NIL. Otherwise, ADDMAPBUFFER causes an error UNABLE TO ALLOCATE PMAP BUFFER.

If *TEMP*=T, the buffers are allocated on a "temporary" basis: allocation takes place via a RESETSAVE whose restoration form will de-allocate the buffers.

(MAPBUFFERCOUNT *ONLYUNLOCKED*) [Function]

Returns the number of buffers currently allocated. If *ONLYUNLOCKED*=T, counts only unlocked buffers; otherwise, counts all buffers. Thus, to insure that at least 3 (unlocked) buffers are allocated, the user could perform (while (LESSP (MAPBUFFERCOUNT T) 3) do (ADDMAPBUFFER NIL T)).

(MAPPAGE *PAGE# FILE* —) [Function]

The primitive function for mapping in pages from *FILE* into the queue of buffers. *PAGE#* is a page number in *FILE*. The value of MAPPAGE is a pointer to the word in memory at which the first word of the page is located, which will always be at a page-boundary.

If *FILE* is NIL, the value of DEFAULTMAPFILE is used.

MAPPAGE searches the buffers to see if the given page for the given file has already been mapped in. If so, it returns the core address to which it was previously mapped. Otherwise, it replaces the previous contents of the least-recently-used buffer with the specified file page. It is important to note that the contents of a given core buffer are not guaranteed across calls to MAPPAGE, unless the page has been locked down via LOCKMAP. MAPPAGE compiles open, and in the case where the desired page is already in the buffer it is quite efficient.

MAPPAGE will allocate an additional buffer if no unlocked buffers are available (and the desired page is not already mapped in).

In Interlisp-10, *FILE* may also be a fork handle (i.e. a value of SUBSYS, page 22.21), in which case the specified page from that fork will be mapped in.

(MAPWORD *FILEADR FILE*) [Function]

Like MAPPAGE, except that it allows the specification of a word-address in *FILE*, not just a page number. MAPWORD determines what page that address is on, maps that page into a buffer (using MAPPAGE), and returns a pointer into the middle of the buffer where the indicated word appears. The rest of the words on the same file page appear at the appropriate word offsets from the value returned by MAPWORD.

(WORDOFFSET *PTR N*) [Function]

If *PTR* is a pointer into a buffer as returned by MAPPAGE or MAPWORD, WORDOFFSET returns a pointer to the *N*th following word. WORDOFFSET compiles open.

(WORDCONTENTS *PTR*) [Function]

Returns the contents of the word at *PTR* as an integer. For example, (WORDCONTENTS (MAPWORD 10 *FILE*)) will return the value stored in word 10 of a (binary) file. WORDCONTENTS compiles open.

(SETWORDCONTENTS *PTR N*) [Function]

Sets the contents of the word pointed to by *PTR* to be the number *N*. Interpreted, SETWORDCONTENTS checks that *PTR* actually is a pointer as returned by MAPPAGE or MAPWORD. SETWORDCONTENTS compiles open with no error checks.

(CLEARMAP *FILE PAGES RELEASE*) [Function]

FILE specifies a file or fork as for MAPPAGE, or it is T. *PAGES* is a single page number or a list of page numbers. CLEARMAP unmaps any of those pages that are currently mapped in, making those buffers available for other mappings. *FILE*=T means all files; *PAGES*=NIL means all pages. Thus (CLEARMAP T) will completely clear the buffers.

Note that CLEARMAP unmaps any pages, whether or not they are currently locked, i.e., CLEARMAP takes precedence over LOCKMAP.

If *RELEASE*=T, then not only will the buffers containing the specified pages be unmapped, but the buffers themselves will be released, i.e. returned to the Interlisp storage manager.

(LOCKMAP *PTR*) [Function]

For those situations in which a program needs prolonged access to a particular file page, LOCKMAP can be used to prevent MAPPAGE from shifting or unmapping the contents of the given core page. *PTR* is a pointer into a mapped page (i.e. a value of MAPWORD or MAPPAGE). LOCKMAP locks the indicated page in core until a corresponding UNLOCKMAP has been performed. If a page has been locked twice, it must be unlocked twice before it is available for reuse. Returns *PTR*.

`(UNLOCKMAP PTR)` [Function]

PTR is a pointer into a mapped page. `UNLOCKMAP` removes the most recent lock for that page.

CHAPTER 15

DWIM

A surprisingly large percentage of the errors made by Interlisp users are of the type that could be corrected by another LISP programmer without any information about the purpose of the program or expression in question, e.g., misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in Interlisp a DWIM facility, short for Do-What-I-Mean. DWIM is called automatically whenever an error occurs in the evaluation of an Interlisp expression. (Currently, DWIM only operates on unbound atoms and undefined function errors.) DWIM then proceeds to try to correct the mistake using the current context of computation plus information about what the user had previously been doing, (and what mistakes he had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last ERRORSET (page 9.15). The following protocol illustrates the operation of DWIM.

For example, suppose the user defines the factorial function (FACT N) as follows:

```
←DEFINEQ((FACT (LAMBDA (N) (COND
((ZEROP N9 1) ((T (ITIMS N (FACCT 8SUB1 N]
(FACT)
←
```

Note that the definition of FACT contains several mistakes: ITIMES and FACT have been misspelled; the 9 in N9 was intended to be a right parenthesis, but the shift key was not depressed; similarly, the 8 in 8SUB1 was intended to be a left parenthesis; and finally, there is an extra left parenthesis in front of the T that begins the final clause in the conditional.

```
←PRETTYPRNT((FACCT]
=PRETTYPRINT
=FACT

(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N9 1)
        ((T (ITIMS N (FACCT 8SUB1 N])
(FACT)
←
```

After defining FACT, the user wishes to look at its definition using PRETTYPRINT, which he unfortunately misspells. Since there is no function PRETTYPRNT in the system, an undefined function error occurs, and DWIM is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by *this* user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRNT meant PRETTYPRINT, notifies the user of this, and calls PRETTYPRINT.

At this point, PRETTYPRINT would normally print (FACCT NOT PRINTABLE) and exit, since FACCT has no definition. Note that this is *not* an Interlisp error condition, so that DWIM would not be called as described above. However, it is obviously not what the user *meant.*

This sort of mistake is corrected by having PRETTYPRINT itself explicitly invoke the spelling corrector portion of DWIM whenever given a function with no EXPR definition. Thus, with the aid of DWIM PRETTYPRINT is able to determine that the user wants to see the definition of the function FACT, and proceeds accordingly.

```
←FACT(3]
N9 [IN FACT] -> N ) ?  YES
[IN FACT] (COND -- ((T --))) ->
             (COND -- (T --))
ITIMS [IN FACT] -> ITIMES
FACCT [IN FACT] -> FACT
8SUB1 [IN FACT] ->  ( SUB1 ?  YES
6
←PP FACT
(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES N (FACT (SUB1 N])
FACT
←
```

The user now calls FACT. During its execution, five errors occur, and DWIM is called five times. At each point, the error is corrected, a message is printed describing the action taken, and the computation is allowed to continue as if no error had occurred. Following the last correction, 6 is printed, the value of (FACT 3). Finally, the user prettyprints the new, now correct, definition of FACT.

In this particular example, the user was shown operating in TRUSTING mode, which gives DWIM carte blanche for most corrections. The user can also operate in CAUTIOUS mode, in which case DWIM will inform him of intended corrections before they are made, and allow the user to approve or disapprove of them. If DWIM was operating in CAUTIOUS mode in the example above, it would proceed as follows:

```
←FACT(3)
N9 [IN FACT] -> N ) ?  YES
U.D.F. T [IN FACT]   FIX?   YES
[IN FACT] (COND -- ((T --))) ->
             (COND -- (T --))
ITIMS [IN FACT] -> ITIMES ?   ...YES
FACCT [IN FACT] -> FACT ?   ...YES
8SUB1 [IN FACT] ->  ( SUB1 ?  NO
U.B.A.
(8SUB1 BROKEN)
:
```

For most corrections, if the user does not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that the user need intervene only when he does not approve. Note that the user responded to the first, second, and fifth questions; DWIM responded for him on the third and fourth.[1]

A great deal of effort has gone into making DWIM "smart", and experience with a large number of users indicates that DWIM works very well; DWIM seldom fails to correct an error the user feels it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done:[2] since an error had occurred, the user would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, the user simply interrupts or aborts the computation, `UNDO`es the DWIM change using `UNDO` (page 8.11), and makes the correction he would have had to make without DWIM. It is this benign quality of DWIM that makes it a valuable part of Interlisp.

(`DWIM` *x*) [Function]

Used to enable/disable DWIM. If *x* is the litatom `C`, DWIM is enabled in `CAUTIOUS` mode, so that DWIM will ask the user before making corrections. If *x* is `T`, DWIM is enabled in `TRUSTING` mode, so DWIM will make most corrections automatically. If *x* is `NIL`, DWIM is disabled. Interlisp initially has DWIM enabled in `CAUTIOUS` mode.

`DWIM` returns `CAUTIOUS`, `TRUSTING` or `NIL`, depending to what mode it has just been put into.

For corrections to expressions typed in by the user for immediate execution,[3] DWIM always acts as though it were in `TRUSTING` mode, i.e., no approval necessary. For certain types of corrections, e.g., run-on spelling corrections, 8-9 errors, etc., DWIM always acts like it was in `CAUTIOUS` mode, and asks for approval. In either case, DWIM always informs the user of its action as described below.

15.1 SPELLING CORRECTION PROTOCOL

One type of error that DWIM can correct is the misspelling of a function or a variable name. When an unbound litatom or undefined function error occurs, DWIM tries to correct the spelling of the bad litatom. If a litatom is found whose spelling is "close" to the offender, DWIM proceeds as follows:

If the correction occurs in the typed-in expression, DWIM prints `=`*CORRECT-SPELLING*^{cr} and continues evaluating the expression. For example:

[1] DWIM uses `ASKUSER` for its interactions with the user (page 6.57). Whenever an interaction is about to take place and the user has typed ahead, `ASKUSER` types several bells to warn the user to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if the user has typed ahead before a DWIM interaction, DWIM will not confuse his type ahead with the answer to its question, nor will his typeahead be lost. The bells are printed by the function `PRINTBELLS`, which can be advised or redefined for specialized applications, e.g. to flash the screen for a display terminal.

[2] Except perhaps if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before the user could interrupt. We have not yet had such an incident occur.

[3] Typed into `LISPX` (see page 8.28).

```
←(SETQ FOO (IPLUSS 1 2))
=IPLUS
3
```

If the correction does not occur in type-in, DWIM prints[4]

BAD-SPELLING [IN *FUNCTION-NAME*] -> *CORRECT-SPELLING*

Then, if DWIM is in TRUSTING mode, it prints a carriage return, makes the correction, and continues the computation. If DWIM is in CAUTIOUS mode, it prints a few spaces and ? and then wait for approval. The user then has six options:

(1) Type Y. DWIM types es, and proceeds with the correction.

(2) Type N. DWIM types o, and does not make the correction.

(3) Type ↑. DWIM does not make the correction, and furthermore guarantees that the error will not cause a break.

(4) Type control-E. For error correction, this has the same effect as typing N.

(5) Do nothing. In this case DWIM waits for DWIMWAIT seconds, and if the user has not responded, DWIM will type ... followed by the default answer.

The default on spelling corrections is determined by the value of the variable FIXSPELLDEFAULT, whose top level value is initially Y.

(6) Type space or carriage-return. In this case DWIM will wait indefinitely. This option is intended for those cases where the user wants to think about his answer, and wants to insure that DWIM does not get "impatient" and answer for him.

The procedure for spelling correction on other than Interlisp errors is analogous. If the correction is being handled as type-in, DWIM prints = followed by the correct spelling, and returns it to the function that called DWIM. Otherwise, DWIM prints the incorrect spelling, followed by the correct spelling. Then, if DWIM if in TRUSTING mode, DWIM prints a carriage-return and returns the correct spelling. Otherwise, DWIM prints a few spaces and a ? and waits for approval. The user can then respond with Y, N, control-E, space, carriage return, or do nothing as described above.

Note that the spelling corrector itself is not ERRORSET protected like the DWIM error correction routines. Therefore, typing N and typing control-E may have different effects when the spelling corrector is called directly. The former simply instructs the spelling corrector to return NIL, and lets the calling function

[4] The appearance of -> is to call attention to the fact that the user's function will be or has been changed.

decide what to do next; the latter causes an error which unwinds to the last ERRORSET, however far back that may be.

15.2 PARENTHESES ERRORS PROTOCOL

When an unbound litatom or undefined error occurs, and the offending litatom contains 8 or 9, DWIM tries to correct errors caused by typing 8 for left parenthesis and 9 for right parenthesis.[5] In these cases, the interaction with the user is similar to that for spelling correction. If the error occurs in type-in, DWIM types =*CORRECTION*^{cr}, and continues evaluating the expression. For example:

```
←(SETQ FOO 8IPLUS 1 2]
= ( IPLUS
3
```

If the correction does not occur in type-in, DWIM prints

BAD-ATOM [IN *FUNCTION-NAME*] -> *CORRECTION* ?

and then waits for approval. The user then has the same six options as for spelling correction, except the waiting time is 3*DWIMWAIT seconds. If the user types Y, DWIM then operates as if it were in TRUSTING mode, i.e., it makes the correction and prints its message.

15.3 U.D.F. T ERRORS PROTOCOL

When an undefined function error occurs, and the offending function is T, DWIM tries to correct certain types of parentheses errors involving a T clause in a conditional. DWIM recognizes errors of the following forms:

(COND --) (T --) The T clause appears outside and immediately following the COND.

(COND -- (-- & (T --))) The T clause appears inside a previous clause.

(COND -- ((T --))) The T clause has an extra pair of parentheses around it.

For U.D.F. T errors that are not one of these three types, DWIM takes no corrective action at all, and the error will occur.

[5] Actually, DWIM uses the value of the variables LPARKEY and RPARKEY to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY are initially 8 and 9 respectively, but they can be reset for other keyboard layouts., e.g., on some terminals left parenthesis is over 9, and right parenthesis is over 0.

If the error occurs in type-in, DWIM simply types T FIXED and makes the correction. Otherwise if DWIM is in TRUSTING mode, DWIM makes the correction and prints the message:

```
[IN FUNCTION-NAME] {BAD-COND} ->
                        {CORRECTED-COND}
```

If DWIM is in CAUTIOUS mode, DWIM prints

```
U.D.F. T
[IN FUNCTION-NAME]    FIX?
```

and waits for approval. The user then has the same options as for spelling corrections and parenthesis errors. If the user types Y or defaults, DWIM makes the correction and prints its message.

Having made the correction, DWIM must then decide how to proceed with the computation. In the first case, (COND --) (T --), DWIM cannot know whether the T clause would have been executed if it had been inside of the COND. Therefore DWIM asks the user CONTINUE WITH T CLAUSE (with a default of YES). If the user types N, DWIM continues with the form after the COND, i.e., the form that originally followed the T clause.

In the second case, (COND -- (-- & (T --))), DWIM has a different problem. After moving the T clause to its proper place, DWIM must return as the value of & as the value of the COND. Since this value is no longer around, DWIM asks the user, OK TO REEVALUATE and then prints the expression corresponding to &. If the user types Y, or defaults, DWIM continues by reevaluating &, otherwise DWIM aborts, and a U.D.F. T error will then occur (even though the COND has in fact been fixed).[6]

In the third case, (COND -- ((T --))), there is no problem with continuation, so no further interaction is necessary.

15.4 DWIM OPERATION

Whenever the interpreter encounters an atomic form with no binding, or a non-atomic form CAR of which is not a function or function object, it calls the function FAULTEVAL. Similarly, when APPLY is given an undefined function, FAULTAPPLY is called. When DWIM is enabled, FAULTEVAL and FAULTAPPLY are redefined to first call the DWIM package, which tries to correct the error. If DWIM cannot decide how to fix the error, or the user disapproves of DWIM's correction (by typing N), or the user types control-E, then FAULTEVAL and FAULTAPPLY cause an error or break.[7]

If DWIM can (and is allowed to) correct the error, it exits by performing RETEVAL of the corrected form, as of the position of the call to FAULTEVAL or FAULTAPPLY. Thus in the example at the beginning of the chapter, when DWIM determined that ITIMS was ITIMES misspelled, DWIM called RETEVAL

[6] If DWIM can determine for itself that the form can safely be reevaluated, it does not consult the user before reevaluating. DWIM can do this if the form is atomic, or CAR of the form is a member of the list OKREEVALST, and each of the arguments can safely be reevaluated. For example, (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on OKREEVALST.

[7] If the user types ↑ to DWIM, DWIM exits by performing (RETEVAL 'FAULTEVAL '(ERROR!)), so that an error will be generated at the position of the call to FAULTEVAL.

with (ITIMES N (FACCT 8SUB1 N)). Since the interpreter uses the value returned by FAULTEVAL exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible; in the above example, DWIM also changed (with RPLACA) the expression (ITIMS N (FACCT 8SUB1 N)) that caused the error. Note that if the user's program had *computed* the form and called EVAL, it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

Error correction in DWIM is divided into three categories: unbound atoms, undefined CAR of form, and undefined function in APPLY. Assuming that the user approves DWIM's corrections, the action taken by DWIM for the various types of errors in each of these categories is summarized below.

15.4.1 DWIM Correction: Unbound Atoms

If DWIM is called as the result of an unbound atom error, it proceeds as follows:

(1) If the first character of the unbound atom is ', DWIM assumes that the user (intentionally) typed '*ATOM* for (QUOTE *ATOM*) and makes the appropriate change. No message is typed, and no approval is requested.

If the unbound atom is just ' itself, DWIM assumes the user wants the *next* expression quoted, e.g., (CONS X '(A B C)) will be changed to (CONS X (QUOTE (A B C))). Again no message will be printed or approval asked. If no expression follows the ', DWIM gives up.[8]

(2) If CLISP (page 16.1) is enabled, and the atom is part of a CLISP construct, the CLISP transformation is performed and the result returned. For example, N-1 is transformed to (SUB1 N), and (··· FOO←3 ···) is transformed into (··· (SETQ FOO 3) ···).

(3) If the atom contains an 8 (actually LPARKEY, see page 15.12), DWIM assumes the 8 was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that the user did not notice the mistake, i.e., that the entire expression was affected by the missing left parenthesis. For example, if the user types (SETQ X (LIST (CONS 8CAR Y) (CDR Z)) Y), the expression will be changed to (SETQ X (LIST (CONS (CAR Y) (CDR Z)) Y)). Note that the 8 does not have to be the first character of the atom: DWIM will handle (CONS X8CAR Y) correctly.

(4) If the atom contains a 9 (actually RPARKEY, see page 15.12), DWIM assumes the 9 was intended to be a right parenthesis and operates as in the case above.

(5) If the atom begins with a 7, the 7 is treated as a '. For example, 7FOO becomes 'FOO, and then (QUOTE FOO).

(6) If the atom is an edit command (a member of EDITCOMSA), and the error occurred in type-in, the effect is the same as though the user typed EDITF(), followed by the atom, i.e., DWIM assumes the user wants to be in the editor editing the last thing he referred to. Thus, if the user defines the function

[8] ' is normally defined as a read-macro character which converts 'FOO to (QUOTE FOO) on input, so DWIM will not see the ' in the case of expressions that are typed-in.

FOO and then types P, he will see =FOO, followed by EDIT, followed by the printout associated with the execution of the P command, followed by *, at which point he can continue editing FOO.

(7) The expressions on DWIMUSERFORMS (see page 15.10) are evaluated in the order that they appear. If any of these expressions returns a non-NIL value, this value is treated as the form to be used to continue the computation, it is evaluated and its value is returned by DWIM.

(8) If the unbound atom occurs in a function, DWIM attempts spelling correction using the LAMBDA and PROG variables of the function as the spelling list.

(9) If the unbound atom occurred in a type-in to a break, DWIM attempts spelling correction using the LAMBDA and PROG variables of the broken function as the spelling list.

(10) Otherwise, DWIM attempts spelling correction using SPELLINGS3 (see page 15.14).

(11) If all of the above fail, DWIM gives up.

15.4.2 Undefined CAR of Form

If DWIM is called as the result of an undefined CAR of form error, it proceeds as follows:

(1) If CAR of the form is T, DWIM assumes a misplaced T clause and operates as described on page 15.5.

(2) If CAR of the form is F/L, DWIM changes the "F/L" to "FUNCTION(LAMBDA". For example, (F/L (Y) (PRINT (CAR Y))) is changed to (FUNCTION (LAMBDA (Y) (PRINT (CAR Y))). No message is printed and no approval requested. If the user omits the variable list, DWIM supplies (X), e.g., (F/L (PRINT (CAR X))) is changed to (FUNCTION (LAMBDA (X) (PRINT (CAR X)))). DWIM determines that the user has supplied the variable list when more than one expression follows F/L, CAR of the first expression is not the name of a function, and every element in the first expression is atomic. For example, DWIM will supply (X) when correcting (F/L (PRINT (CDR X)) (PRINT (CAR X))).

(3) If CAR of the form is a CLISP word (IF, FOR, DO, FETCH, etc.), the indicated CLISP transformation is performed, and the result is returned as the corrected form. See page 16.1.

(4) If CAR of the form has a function definition, DWIM attempts spelling correction on CAR of the definition using as spelling list the value of LAMBDASPLST, initially (LAMBDA NLAMBDA).

(5) If CAR of the form has an EXPR or CODE property, DWIM prints *CAR-OF-FORM* UNSAVED, performs an UNSAVEDEF, and continues. No approval is requested.

(6) If CAR of the form has a FILEDEF property, the definition is loaded from a file.[9] If the value of the property is atomic, the entire file is to be loaded. If the value is a list, CAR is the name of the file and CDR the relevant functions, and LOADFNS will be used. For both cases, *LDFLG* will be SYSLOAD (see page 11.4). DWIM uses FINDFILE (page 15.20), so that the file can be on any of the directories on DIRECTORIES, initially (NIL NEWLISP LISP LISPUSERS). If the file is found, DWIM types SHALL I LOAD followed by the file name or list of functions. If the user approves, DWIM loads the function(s) or file, and continues the computation.

[9] except when DWIMIFYing.

(7) If CLISP is enabled, and CAR of the form is part of a CLISP construct, the indicated transformation is performed, e.g., (N←N-1) becomes (SETQ N (SUB1 N)).

(8) If CAR of the form contains an 8, DWIM assumes a left parenthesis was intended e.g., (CONS8CAR X).

(9) If CAR of the form contains a 9, DWIM assumes a right parenthesis was intended.

(10) If CAR of the form is a list, DWIM attempts spelling correction on CAAR of the form using LAMBDASPLST as spelling list. If successful, DWIM returns the corrected expression itself.

(11) If CAR of the form is a small number, and the error occurred in type-in, DWIM assumes the form is really an edit command and operates as described for unbound atom errors above.

(12) If CAR of the form is an edit command (a member of EDITCOMSL), DWIM operates as in the previous case.

(13) The expressions on DWIMUSERFORMS are evaluated in the order they appear. If any returns a non-NIL value, this value is treated as the corrected form, it is evaluated, and DWIM returns its value.

(14) Otherwise, DWIM attempts spelling correction using SPELLINGS2 as the spelling list (see page 15.14). When DWIMIFYing, DWIM also attemps spelling correction on function names not defined but previously encountered, using NOFIXFNSLST as a spelling list (see page 16.16).

(15) If all of the above fail, DWIM gives up.

15.4.3 Undefined Function in APPLY

If DWIM is called as the result of an undefined function in APPLY error, it proceeds as follows:

(1) If the function has a definition, DWIM attempts spelling correction on CAR of the definition using LAMBDASPLST as spelling list.

(2) If the function has an EXPR or CODE property, DWIM prints *FN* UNSAVED, performs an UNSAVEDEF and continues. No approval is requested.

(3) If the function has a property FILEDEF, DWIM proceeds as in case 6 of undefined CAR of form.

(4) If the error resulted from type-in, and CLISP is enabled, and the function name contains a CLISP operator, DWIM performs the indicated transformation, e.g., the user types FOO←(APPEND FIE FUM).

(5) If the function name contains an 8, DWIM assumes a left parenthesis was intended, e.g., EDIT8FOO].

(6) If the "function" is a list, DWIM attempts spelling correction on CAR of the list using LAMBDASPLST as spelling list.

(7) If the function is a number and the error occurred in type-in, DWIM assumes the function is an edit command, and operates as described in case 6 of unbound atoms, e.g., the user types (on one line) 3 -1 P.

(8) If the function is the name of an edit command (on either EDITCOMSA or EDITCOMSL), DWIM operates as in the previous case, e.g., user types F COND.

(9) The expressions on DWIMUSERFORMS are evaluated in the order they appear, and if any returns a non-NIL value, this value is treated as the function used to continue the computation, i.e., it will be applied to its arguments.

(10) DWIM attempts spelling correction using SPELLINGS1 as the spelling list.

(11) DWIM attempts spelling correction using SPELLINGS2 as the spelling list.

(12) If all fail, DWIM gives up.

15.5 DWIMUSERFORMS

The variable DWIMUSERFORMS provides a convenient way of adding to the transformations that DWIM performs. For example, the user might want to change atoms of the form $X to (QA4LOOKUP X). Before attempting spelling correction, but after performing other transformations (F/L, 8, 9, CLISP, etc.), DWIM evaluates the expressions on DWIMUSERFORMS in the order they appear. If any expression returns a non-NIL value, this value is treated as the transformed form to be used. If DWIM was called from FAULTEVAL, this form is evaluated and the resulting value is returned as the value of FAULTEVAL. If DWIM is called from FAULTAPPLY, this form is treated as a function to be applied to FAULTARGS, and the resulting value is returned as the value of FAULTAPPLY. If all of the expressions on DWIMUSERFORMS return NIL, DWIM proceeds as though DWIMUSERFORMS=NIL, and attempts spelling correction. Note that DWIM simply takes the value and returns it; the expressions on DWIMUSERFORMS are responsible for making any modifications to the original expression.[10]

In order for an expression on DWIMUSERFORMS to be able to be effective, it needs to know various things about the context of the error. Therefore, several of DWIM's internal variables have been made SPECVARS (see page 12.4) and are therefore "visible" to DWIMUSERFORMS. Below are a list of those variables that may be useful.

FAULTX [Variable]

For unbound atom and undefined car of form errors, FAULTX is the atom or form. For undefined function in APPLY errors, FAULTX is the name of the function.

FAULTARGS [Variable]

For undefined function in APPLY errors, FAULTARGS is the list of arguments. FAULTARGS may be modified or reset by expressions on DWIMUSERFORMS.

FAULTAPPLYFLG [Variable]

Value is T for undefined function in APPLY errors; NIL otherwise. The value of FAULTAPPLYFLG *after* an expression on DWIMUSERFORMS returns a non-NIL value determines how the latter value is to be treated. Following an undefined function in APPLY error, if an expression on DWIMUSERFORMS sets FAULTAPPLYFLG to NIL, the value returned is treated as a form to be evaluated, rather than a function to be applied.

[10] The expressions on DWIMUSERFORMS should make the transformation permanent, either by associating it with FAULTX via CLISPTRAN, or by physically smashing FAULTX.

FAULTAPPLYFLG is necessary to distinguish between unbound atom and undefined function in APPLY errors, since FAULTARGS may be NIL and FAULTX atomic in both cases.

TAIL [Variable]

For unbound atom errors, TAIL is the tail of the expression CAR of which is the unbound atom. DWIMUSERFORMS expression can replace the atom by another expression by performing (/RPLACA TAIL *EXPR*)

PARENT [Variable]

For unbound atom errors, PARENT is the form in which the unbound atom appears. TAIL is a tail of PARENT.

TYPE-IN? [Variable]

True if the error occurred in type-in.

FAULTFN [Variable]

Name of the function in which error occurred. FAULTFN is TYPE-IN when the error occurred in type-in, and EVAL or APPLY when the error occurred under an explicit call to EVAL or APPLY.

DWIMIFYFLG [Variable]

True if the error was encountered while DWIMIFYing (as opposed to happening while running a program).

EXPR [Variable]

Definition of FAULTFN, or argument to EVAL, i.e., the superform in which the error occurs.

The initial value of DWIMUSERFORMS is ((MACROTRAN) (DWIMLOADFNS?)). MACROTRAN is a package for running interpreted programs containing ASSEMBLE statements or calls to "functions" defined only by MACRO properties (see page 5.19). DWIMLOADFNS? is a function for automatically loading functions from files. If DWIMLOADFNSFLG is T (its initial value), and CAR of the form is the name of a function, and the function is contained on a file that has been noticed by the file package, the function is loaded, and the computation continues.

15.6 DWIM FUNCTIONS AND VARIABLES

DWIMWAIT [Variable]

Value is the number of seconds that DWIM will wait before it assumes that the user is not going to respond to a question and uses the default response FIXSPELLDEFAULT.

DWIM operates by dismissing for 250 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until DWIMWAIT seconds have elapsed. Thus, there will be a delay of at most 1/4 second before DWIM responds to the user's answer.

FIXSPELLDEFAULT [Variable]

If approval is requested for a spelling correction, and user does not respond, defaults to value of FIXSPELLDEFAULT, initially Y. FIXSPELLDEFAULT is rebound to N when DWIMIFYing.

ADDSPELLFLG [Variable]

If NIL, suppresses calls to ADDSPELL. Initially T.

NOSPELLFLG [Variable]

If T, suppresses *all* spelling correction. If some other non-NIL value, suppresses spelling correction in programs but not type-in. NOSPELLFLG is initially NIL. It is rebound to T when compiling from a file.

RUNONFLG [Variable]

If NIL, suppresses run-on spelling corrections. Initially T.

DWIMLOADFNSFLG [Variable]

If T, tells DWIM that when it encounters a call to an undefined function contained on a file that has been noticed by the file package, to simply load the function. DWIMLOADFNSFLG is initially T. See page 15.11.

LPARKEY [Variable]
RPARKEY [Variable]

DWIM uses the value of the variables LPARKEY and RPARKEY (initially 8 and 9 respectively) to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY can be reset for other keyboard layouts., For example, on some terminals left parenthesis is over 9, and right parenthesis is over 0.

OKREEVALST [Variable]

The value of OKREEVALST is a list of functions that DWIM can safely reevaluate. If a form is atomic, or CAR of the form is a member of OKREEVALST, and each of the arguments can safely be reevaluated, then the form can be safely reevaluated. For example, (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on OKREEVALST.

DWIMFLG [Variable]

DWIMFLG=NIL, all DWIM operations are disabled. (DWIM 'C) and (DWIM T) set DWIMFLG to T; (DWIM NIL) sets DWIMFLG to NIL.

APPROVEFLG [Variable]

APPROVEFLG=T if DWIM should ask the user for approval before making a correction that will modify the definition of one of his functions; NIL otherwise.

When DWIM is put into CAUTIOUS mode with (DWIM 'C), APPROVEFLG is set to T; for TRUSTING mode, APPROVEFLG is set to NIL.

LAMBDASPLST [Variable]

DWIM uses the value of LAMBDASPLST as the spelling list when correcting "bad" function definitions. Initially (LAMBDA NLAMBDA). The user may wish to add to LAMBDASPLST if he elects to define new "function types" via an appropriate DWIMUSERFORMS entry. For example, the QLAMBDAs of SRI's QLISP are handled in this way.

15.7 SPELLING CORRECTION

The spelling corrector is given as arguments a misspelled word (word means literal atom), a spelling list (a list of words), and a number: *XWORD*, *SPLST*, and *REL* respectively. Its task is to find that word on *SPLST* which is closest to *XWORD*, in the sense described below. This word is called a *respelling* of *XWORD*. *REL* specifies the minimum "closeness" between *XWORD* and a respelling. If the spelling corrector cannot find a word on *SPLST* closer to *XWORD* than *REL*, or if it finds two or more words equally close, its value is NIL, otherwise its value is the respelling. The spelling corrector can also be given an optional functional argument, *FN*, to be used for selecting out a subset of *SPLST*, i.e., only those members of *SPLST* that satisfy *FN* will be considered as possible respellings.

The exact algorithm for computing the spelling metric is described later, but briefly "closeness" is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word. For example, PRTTYPRNT is "closer" to PRETTYPRINT than CS is to CONS even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down *SPLST*, and computing the closeness between each word and *XWORD*, and keeping a list of those that are closest. Certain differences between words are not counted as disagreements, for example a single transposition, e.g., CONS to CNOS, or a doubled letter, e.g., CONS to CONSS, etc. In the event that the spelling corrector finds a word on *SPLST* with *no* disagreements, it will stop searching and return this word as the respelling. Otherwise, the spelling corrector continues through the entire spelling list. Then if it has found one and only one "closest" word, it returns this word as the respelling. For example, if *XWORD* is VONS, the spelling corrector will probably return CONS as the respelling. However, if *XWORD* is CONZ, the spelling corrector will not be able to return a respelling, since CONZ is equally close to both CONS and COND. If the spelling corrector finds an acceptable respelling, it interacts with the user as described earlier.

In the special case that the misspelled word contains one or more $s (<esc>s, alt-mode on some terminals), the spelling corrector searches for those words on *SPLST* that match *XWORD*, where a $ can match any number of characters (including 0), e.g., FOO$ matches FOO1 and FOO, but not NEWFOO. FOO matches all three. Both completion and correction may be involved, e.g. RPETTY$ will match PRETTYPRINT, with one mistake. The entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints AMBIGUOUS, and returns NIL. For example, CON$ would be ambiguous if both CONS and COND were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with the user as described earlier.

For both spelling correction and spelling completion, regardless of whether or not the user approves of the spelling corrector's choice, the respelling is moved to the front of *SPLST*. Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

15.7.1 Synonyms

Spelling lists also provide a way of defining synonyms for a particular context. If a dotted pair appears on a spelling list (instead of just an atom), CAR is interpreted as the correct spelling of the misspelled word, and CDR as the antecedent for that word. If CAR is *identical* with the misspelled word, the antecedent is returned without any interaction or approval being necessary. If the misspelled word *corrects* to CAR of the dotted pair, the usual interaction and approval will take place, and then the antecedent, i.e., CDR of the dotted pair, is returned. For example, the user could make IFLG synonymous with CLISPIFTRANFLG by adding (IFLG . CLISPIFTRANFLG) to SPELLINGS3, the spelling list for unbound atoms. Similarly, the user could make OTHERWISE mean the same as ELSEIF by adding (OTHERWISE . ELSEIF) to CLISPIFWORDSPLST, or make L be synonymous with LAMBDA by adding (L . LAMBDA) to LAMBDASPLST. Note that L could also be used as a variable without confusion, since the association of L with LAMBDA occurs only in the appropriate context.

15.7.2 Spelling Lists

Any list of atoms can be used as a spelling list, e.g., BROKENFNS, FILELST, etc. Various system packages have their own spellings lists, e.g., LISPXCOMS, CLISPFORWORDSPLST, EDITCOMSA, etc. These are documented under their corresponding sections, and are also indexed under "spelling lists." In addition to these spelling lists, the system maintains, i.e., automatically adds to, and occasionally prunes, four lists used solely for spelling correction: SPELLINGS1, SPELLINGS2, SPELLINGS3, and USERWORDS. These spelling lists are maintained *only* when ADDSPELLFLG is non-NIL. ADDSPELLFLG is initially T.

SPELLINGS1 [Variable]

SPELLINGS1 is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g., EDTIF(FOO). SPELLINGS1 is initialized to contain DEFINEQ, BREAK, MAKEFILE, EDITF, TCOMPL, LOAD, etc. Whenever LISPX is given an input in apply format, i.e., a function and arguments, the name of the function is added to SPELLINGS1 if the function has a definition.

For example, typing CALLS(EDITF) will cause CALLS to be added to SPELLINGS1. Thus if the user typed CALLS(EDITF) and later typed CALLLS(EDITV), since SPELLINGS1 would then contain CALLS, DWIM would be successful in correcting CALLLS to CALLS.

SPELLINGS2 [Variable]

SPELLINGS2 is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as ADD1, APPEND, COND, CONS, GO, LIST, NCONC, PRINT, PROG, RETURN, SETQ, etc. Whenever LISPX is given a non-atomic form, the name of the function is added to SPELLINGS2. For example, typing (RETFROM (STKPOS (QUOTE FOO) 2)) to a break would add RETFROM to SPELLINGS2. Function names are also added to SPELLINGS2 by DEFINE, DEFINEQ, LOAD (when loading compiled code), UNSAVEDEF, EDITF, and PRETTYPRINT.

SPELLINGS3 [Variable]

SPELLINGS3 is a list of words used for spelling correction on all unbound atoms. SPELLINGS3 is initialized to EDITMACROS, BREAKMACROS, BROKENFNS, and ADVISEDFNS. Whenever LISPX is given an atom to evaluate, the name of the atom is added to SPELLINGS3 if the atom has a value. Atoms are also added to SPELLINGS3 whenever they are edited by EDITV, and whenever they are set via RPAQ or RPAQQ. For example, when a file is loaded, all of the variables set in the file are added to SPELLINGS3. Atoms are also added to SPELLINGS3 when they are set by a LISPX input, e.g., typing (SETQ FOO (REVERSE (SETQ FIE

...))) will add both FOO and FIE to SPELLINGS3.

USERWORDS [Variable]

USERWORDS is a list containing both functions and variables that the user has *referred* to, e.g., by breaking or editing. USERWORDS is used for spelling correction by ARGLIST, UNSAVEDEF, PRETTYPRINT, BREAK, EDITF, ADVISE, etc. USERWORDS is initially NIL. Function names are added to it by DEFINE, DEFINEQ, LOAD, (when loading compiled code, or loading exprs to property lists) UNSAVEDEF, EDITF, EDITV, EDITP, PRETTYPRINT, etc. Variable names are added to USERWORDS at the same time as they are added to SPELLINGS3. In addition, the variable LASTWORD is always set to the last word added to USERWORDS, i.e., the last function or variable referred to by the user, and the respelling of NIL is defined to be the value of LASTWORD. Thus, if the user has just defined a function, he can then edit it by simply typing EDITF(), or prettyprint it by typing PP().

Each of the above four spelling lists are divided into two sections separated by a special marker. The first section contains the "permanent" words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section (except that functions added to SPELLINGS1 or SPELLINGS2 by LISPX are always added to the end of the permanent section. If the word is already in the temporary section, it is moved to the front of that section; if the word is in the permanent section, no action is taken. If the length of the temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e., deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling correction time. Since the spelling corrector usually moves each word selected as a respelling to the front of its spelling list, the word is thereby moved into the permanent section. Thus once a word is misspelled and corrected, it is considered important and will never be forgotten.

#SPELLINGS1 [Variable]
#SPELLINGS2 [Variable]
#SPELLINGS3 [Variable]
#USERWORDS [Variable]

The maximum length of the temporary section for SPELLINGS1, SPELLINGS2, SPELLINGS3 and USERWORDS is given by the value of #SPELLINGS1, #SPELLINGS2, #SPELLINGS3, and #USERWORDS, initialized to 30, 30, 30, and 60 respectively.

Users can alter these values to modify the performance behavior of spelling correction.

15.7.3 Generators for Spelling Correction

For some applications, it is more convenient to *generate* candidates for a respelling one by one, rather than construct a complete list of all possible candidates, e.g., spelling correction involving a large directory of files, or a natural language data base. For these purposes, *SPLST* can be an array (of any size). The first element of this array is the generator function, which is called with the array itself as its argument. Thus the function can use the remainder of the array to store "state" information, e.g., the last position on a file, a pointer into a data structure, etc. The value returned by the function is the next candidate for respelling. If NIL is returned, the spelling "list" is considered to be exhausted, and the closest match is returned. If a candidate is found with no disagreements, it is returned immediately without waiting for the "list" to exhaust.

SPLST can also be a generator, i.e. the value of the function GENERATOR (page 7.13). The generator *SPLST* will be started up whenever the spelling corrector needs the next candidate, and it should return candidates via the function PRODUCE. For example, the following could be used as a "spelling list" which effectively contains all functions in the system:

```
[GENERATOR
   (MAPATOMS (FUNCTION (LAMBDA (X) (if (GETD X) then (PRODUCE X]
```

15.7.4 Spelling Corrector Algorithm

The basic philosophy of DWIM spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, CONS and CONX differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify a relative agreement of 70, so that a single substitution error is permitted in words of four characters or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, AND and NAD have a relative agreement of 100. Calls to the spelling corrector from DWIM use the value of FIXSPELLREL, which is initially 70. Note that by setting FIXSPELLREL to 100, only spelling corrections with "zero" mistakes, will be considered, e.g., transpositions, double characters, etc.

The central function of the spelling corrector is CHOOZ. CHOOZ takes as arguments: a word, a minimum relative agreement, a spelling list, and an optional functional argument, *XWORD*, *REL*, *SPLST*, and *FN* respectively.

CHOOZ proceeds down *SPLST* examining each word. Words not satisfying *FN* (if *FN* is non-NIL), or those obviously too long or too short to be sufficiently close to *XWORD* are immediately rejected. For example, if *REL*=70, and *XWORD* is 5 characters long, words longer than 7 characters will be rejected.

Special treatment is necessary for words shorter than *XWORD*, since doubled letters are not counted as disagreements. For example, CONNSSS and CONS have a relative agreement of 100. (keyboard bounce on many different kinds of keyboards actually produce this sort of stuttering.) CHOOZ handles this by counting the number of doubled characters in *XWORD* before it begins scanning *SPLST*, and taking this into account when deciding whether to reject shorter words.

If *TWORD*, the current word on *SPLST*, is not rejected, CHOOZ computes the number of disagreements between it and *XWORD* by calling a subfunction, SKOR.

SKOR operates by scanning both words from left to right one character at a time. SKOR operates on the list of character codes for each word. This list is computed by CHOOZ before calling SKOR. Characters are considered to agree if they are the same characters; or appear on the same key (i.e., a shift mistake), for example, * agrees with :, 1 with !, etc.; or if the character in *XWORD* is a lower case version of the character in *TWORD*. Characters that agree are discarded, and the SKORing continues on the rest of the characters in *XWORD* and *TWORD*.

If the first character in *XWORD* and *TWORD* do *not* agree, SKOR checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is counted as a tranposition; a displacement by more than two positions is counted as a disagreement.In either case, both characters are now considered as accounted for and are discarded, and SKORing continues.

If the first character in *XWORD* and *TWORD* do not agree, and neither agree with previously unaccounted-for characters, and *TWORD* has more characters remaining than *XWORD*, SKOR removes and saves the first character of *TWORD*, and continues by comparing the rest of *TWORD* with *XWORD* as described above. If *TWORD* has the same or fewer characters remaining than *XWORD*, the procedure is the same except that the character is removed from *XWORD*. In this case, a special check is first made to see if that character is equal to the *previous* character in *XWORD*, or to the *next* character in *XWORD*, i.e., a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement. In this case, the "length" of *XWORD* is also decremented. Otherwise making *XWORD* sufficiently long by adding double characters would make it be arbitrarily close to *TWORD*, e.g., XXXXXX would correct to PP.

When SKOR has finished processing both *XWORD* and *TWORD* in this fashion, the value of SKOR is the number of unaccounted-for characters, plus the number of disagreements, plus the number of tranpositions, with two qualifications: (1) if both *XWORD* and *TWORD* have a character unaccounted-for in the same position, the two characters are counted only once, i.e., substitution errors count as only one disagreement, not two; and (2) if there are no unaccounted-for characters and no disagreements, transpositions are not counted. This permits spelling correction on very short words, such as edit commands, e.g., XRT->XTR. Transpositions are also not counted when FASTYPEFLG=T, for example, IPULX and IPLUS will be in 80% agreement with FASTYPEFLG=T, only 60% with FASTYPEFLG=NIL. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine tranpositions and other mistakes in a single word, and therefore can use more conservative metric. FASTYPEFLG is initially NIL.

15.7.5 Spelling Corrector Functions and Variables

(ADDSPELL *X SPLST N*) [Function]

Adds *X* to one of the four spelling lists as follows:

If *X* is already on the spelling list, and in its temporary section, ADDSPELL moves *X* to the front of that section.

If *SPLST*=NIL, adds *X* to USERWORDS and to SPELLINGS2. Used by DEFINEQ.

If *SPLST*=0, adds *X* to USERWORDS. Used by LOAD when loading EXPRs to property lists.

If *SPLST*=1, adds *X* to SPELLINGS1 (at end of permanent section). Used by LISPX.

If *SPLST*=2, adds *X* to SPELLINGS2 (at end of permanent section). Used by LISPX.

If *SPLST*=3, adds *X* to USERWORDS and SPELLINGS3.

SPLST can also be a spelling list, in which case *N* is the (optional) length of the temporary section.

ADDSPELL sets LASTWORD to *X* when *SPLST*=NIL, 0 or 3.

If *x* is not a literal atom, ADDSPELL takes no action.

Note that the various systems calls to ADDSPELL, e.g. from DEFINE, EDITF, LOAD, etc., can all be suppressed by setting or binding ADDSPELLFLG to NIL (page 15.12).

(MISSPELLED? *XWORD REL SPLST FLG TAIL FN*) [Function]

If *XWORD*=NIL or $ (<esc>), MISSPELLED? prints = followed by the value of LASTWORD, and returns this as the respelling, without asking for approval. Otherwise, MISSPELLED? checks to see if *XWORD* is really misspelled, i.e., if *FN* applied to *XWORD* is true, or *XWORD* is already contained on *SPLST*. In this case, MISSPELLED? simply returns *XWORD*. Otherwise MISSPELLED? computes and returns (FIXSPELL *XWORD REL SPLST FLG TAIL FN*).

(FIXSPELL *XWORD REL SPLST FLG TAIL FN TIEFLG DONTMOVETOPFLG* — —) [Function]

The value of FIXSPELL is either the respelling of *XWORD* or NIL. If for some reason *XWORD* itself is on *SPLST*, then FIXSPELL aborts and calls ERROR!. If there is a possibility that *XWORD* is spelled correctly, MISSPELLED? should be used instead of FIXSPELL. FIXSPELL performs all of the interactions described earlier, including requesting user approval if necessary.

If *XWORD*=NIL or $ (<esc>), the respelling is the value of LASTWORD, and no approval is requested.

If *XWORD* contains lowercase characters, and the corresponding uppercase word is correct, i.e. on *SPLST* or satisfies *FN*, the uppercase word is returned and no interaction is performed.

If *REL*=NIL, defaults to the value of FIXSPELLREL (initially 70).

If *FLG*=NIL, the correction is handled in type-in mode, i.e., approval is never requested, and *XWORD* is not typed. If *FLG*=T, *XWORD* is typed (before the =) and approval is requested if APPROVEFLG=T. If *FLG*=NO-MESSAGE, the correction is returned with no further processing. In this case, a run-on correction will be returned as a dotted pair of the two parts of the word, and a synonym correction as a list of the form (*WORD1 WORD2*), where *WORD1* is (the corrected version of) *XWORD*, and *WORD2* is the synonym. Note that the effect of the function CHOOZ can be obtained by calling FIXSPELL with *FLG*=NO-MESSAGE.

If *TAIL* is not NIL, and the correction is successful, CAR of *TAIL* is replaced by the respelling (using /RPLACA). In addition, FIXSPELL will correct misspellings caused by running two words together.[11] In this case, CAR of *TAIL* is replaced by the two words, and the value of FIXSPELL is the first one. For example, if FIXSPELL is called to correct the edit command (MOVE TO AFTERCOND 3 2) with *TAIL*=(AFTERCOND 3 2), *TAIL* would be changed to (AFTER COND

[11] In this case, user approval is always requested. In addition, if the first word contains either fewer than 3 characters, or fewer characters than the second word, the default will be N. 'Run-on' spelling corrections can be suppressed by setting the variable RUNONFLG to NIL (initially T).

2 3), and FIXSPELL would return AFTER (subject to user approval where necessary).[12]

If *TIEFLG*=NIL and a tie occurs, i.e., more than one word on *SPLST* is found with the same degree of "closeness", FIXSPELL returns NIL, i.e., no correction. If *TIEFLG*=PICKONE and a tie occurs, the first word is taken as the correct spelling. If *TIEFLG*=LIST, the value of FIXSPELL is a list of the respellings (even if there is only one), and FIXSPELL will not perform any interaction with the user, nor modify *TAIL*, the idea being that the calling program will handle those tasks. Similarly, if *TIEFLG*=EVERYTHING, a list of all candidates whose degree of closeness is above *REL* will be returned, regardless of whether some are better than others. No interaction will be performed.

If *DONTMOVETOPFLG*=T and a correction occurs, it will *not* be moved to the front of the spelling list.

The time required for a call to FIXSPELL with a spelling list of length 60 when the entire list must be searched is .5 seconds. If FIXSPELL determines that the first word on the spelling list is the respelling and does not need to search any further, the time required is .02 seconds. In other words, the time required is proportional to the number of words with which *XWORD* is compared, with the time for one comparison, i.e., one call to SKOR takes roughly .01 seconds (varies slightly with the number of characters in the words being compared).

(FNCHECK *FN NOERRORFLG SPELLFLG PROPFLG TAIL*) [Function]

The task of FNCHECK is to check whether *FN* is the name of a function and if not, to correct its spelling. If *FN* is the name of a function or spelling correction is successful, FNCHECK adds the (corrected) name of the function to USERWORDS using ADDSPELL, and returns it as its value.

Since FNCHECK is called by many low level functions such as ARGLIST, UNSAVEDEF, etc., spelling correction only takes place when DWIMFLG=T, so that these functions can operate in a small Interlisp system which does not contain DWIM.

NOERRORFLG informs FNCHECK whether or not the calling function wants to handle the unsuccessful case: if *NOERRORFLG* is T, FNCHECK simply returns NIL, otherwise it prints fn NOT A FUNCTION and generates a non-breaking error.

If *FN* does not have a definition, but does have an EXPR property, then spelling correction is not attempted. Instead, if *PROPFLG*=T, *FN* is considered to be the name of a function, and is returned. If *PROPFLG*=NIL, *FN* is *not* considered to be the name of a function, and NIL is returned or an error generated, depending on the value of *NOERRORFLG*.

FNCHECK calls MISSPELLED? to perform spelling correction, so that if *FN*=NIL, the value of LASTWORD will be returned. *SPELLFLG* corresponds to MISSPELLED?'s

[12] If *TAIL*=T, FIXSPELL will also perform run-on corrections, returning a dotted pair of the two words in the event the correction is of this type.

fourth argument, *FLG*. If *SPELLFLG*=T, approval will be asked if DWIM was enabled in CAUTIOUS mode, i.e., if APPROVEFLG=T. *TAIL* corresponds to the fifth argument to MISSPELLED?.

FNCHECK is currently used by ARGLIST, UNSAVEDEF, PRETTYPRINT, BREAK0, BREAKIN, ADVISE, CALLS, and EDITA. For example, BREAK0 calls FNCHECK with *NOERRORFLG*=T since if FNCHECK cannot produce a function, BREAK0 wants to define a dummy one. CALLS however calls FNCHECK with *NOERRORFLG*=NIL, since it cannot operate without a function.

Many other system functions call MISSPELLED? or FIXSPELL directly. For example, BREAK1 calls FIXSPELL on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, LISPX calls FIXSPELL on atomic inputs using a list of all LISPX commands. When UNBREAK is given the name of a function that is not broken, it calls FIXSPELL with two different spelling lists, first with BROKENFNS, and if that fails, with USERWORDS. MAKEFILE calls MISSPELLED? using FILELST as a spelling list. Finally, LOAD, BCOMPL, BRECOMPILE, TCOMPL, and RECOMPILE all call MISSPELLED? if their input file(s) won't open.

(SPELLFILE *FILE NOPRINTFLG NSFLG DIRLST*) [Function]

If *FILE* does not have a directory field, SPELLFILE looks on the directories given by the value of DIRECTORIES, initially (NIL LISP). (NIL corresponds to login directory.) This correction will not require user approval, (but SPELLFILE will indicate the correction in the usual way, by printing = followed by the new file name). Otherwise, SPELLFILE attempts spelling correction against the files in the directory. In this case, user approval will be requested (except if *NOPRINTFLG*=T, see below). Returns corrected file, if any, otherwise NIL.

If *NOPRINTFLG*=T, SPELLFILE does not do any printing, nor ask for approval.

If *NSFLG*=T (or NOSPELLFLG=T), no spelling correction is attempted, though searching through DIRECTORIES will still be performed.

If *DIRLST* is non-NIL, it is used instead of the value of DIRECTORIES.

ERRORTYPELST (page 9.16) is initially

```
((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG)))
```

This causes SPELLFILE to be called in case of a FILE NOT FOUND error. If the variable NOFILESPELLFLG is T (its initial value), then spelling correction is not done on the file name, but DIRECTORIES is still searched. If SPELLFILE is successful, the operation will be reexecuted with the new (corrected) file name.

(FINDFILE *FILE NSFLG DIRLST*) [Function]

If *FILE* is not the name of a file, calls SPELLFILE specifying no interaction or printing. FINDFILE could be defined as:

```
(if (INFILEP FILE)
  else (SPELLFILE FILE T NSFLG DIRLST))
```

CHAPTER 16

CLISP

The syntax of LISP is very simple, in the sense that it can be described concisely, but not in the sense that LISP programs are easy to read or write! This simplicity of syntax is achieved by, and at the expense of, extensive use of explicit structuring, namely grouping through parenthesization. Unlike many languages, there are no reserved words in LISP such as `IF`, `THEN`, `FOR`, `DO`, etc., nor reserved characters like +, -, =, ←, etc. The only special characters are left and right parentheses and period, which are used for indicating structure, and space and end-of-line, which are used for delimiting identifiers. This eliminates entirely the need for parsers and precedence rules in the LISP interpreter and compiler, and thereby makes program manipulation of LISP programs straightforward. In other words, a program that "looks at" other LISP programs does not need to incorporate a lot of syntactic information. For example, a LISP interpreter can be written in one or two pages of LISP code. It is for this reason that LISP is by far the most suitable, and frequently used, programming language for writing programs that deal with other programs as data, e.g., programs that analyze, modify, or construct other programs.

However, it is precisely this same simplicity of syntax that makes LISP programs difficult to read and write (especially for beginners). 'Pushing down' is something programs do very well, and people do poorly. As an example, consider the following two "equivalent" sentences:

"The rat that the cat that the dog that I owned chased caught ate the cheese."

versus

"I own the dog that chased the cat that caught the rat that ate the cheese."

Natural language contains many linguistic devices such as that illustrated in the second sentence above for minimizing embedding, because embedded sentences are more difficult to grasp and understand than equivalent non-embedded ones (even if the latter sentences are somewhat longer). Similarly, most high level programming languages offer syntactic devices for reducing apparent depth and complexity of a program: the reserved words and infix operators used in ALGOL-like languages simultaneously delimit operands and operations, and also convey meaning to the programmer. They are far more intuitive than parentheses. In fact, since LISP uses parentheses (i.e., lists) for almost all syntactic forms, there is very little information contained in the parentheses for the person reading a LISP program, and so the parentheses tend mostly to be ignored: the meaning of a particular LISP expression for people is found almost entirely in the *words*, not in the structure. For example, the expression

```
(COND (EQ N 0) 1) (T TIMES N FACTORIAL ((SUB1 N)))
```

is recognizable as factorial even though there are five misplaced or missing parentheses. Grouping words together in parentheses is done more for LISP's benefit, than for the programmer's.

CLISP is designed to make Interlisp programs easier to read and write by permitting the user to employ various infix operators, `IF` statements (page 4.4), and iterative statements (page 4.5), which are automatically converted to equivalent Interlisp expressions when they are first interpreted. For example, factorial could be written in CLISP:

```
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1))
```

Note that this expression would become an equivalent COND after it had been interpreted once, so that programs that might have to analyze or otherwise process this expression could take advantage of the simple syntax.

There have been similar efforts in other LISP systems. CLISP differs from these in that it does not attempt to *replace* the LISP syntax so much as to *augment* it. In fact, one of the principal criteria in the design of CLISP was that users be able to freely intermix LISP and CLISP without having to identify which is which. Users can write programs, or type in expressions for evaluation, in LISP, CLISP, or a mixture of both. In this way, users do not have to learn a whole new language and syntax in order to be able to use selected facilities of CLISP when and where they find them useful.

CLISP is implemented via the error correction machinery in Interlisp (see page 15.1). Thus, any expression that is well-formed from Interlisp's standpoint will never be seen by CLISP (i.e., if the user defined a function IF, he would effectively turn off that part of CLISP). This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the Interlisp interpreter does not "know" about CLISP at all. It operates as before, and when an erroneous form is encountered, the interpreter calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer which contains CLISP. If the expression in question turns out to be a CLISP construct, the equivalent Interlisp form is returned to the interpreter. In addition, the original CLISP expression, is modified so that it *becomes* the correctly translated Interlisp form. In this way, the analysis and translation are done only once.

Integrating CLISP into the Interlisp system (instead of, for example, implementing it as a separate preprocessor) makes possible Do-What-I-Mean features for CLISP constructs as well as for pure LISP expressions. For example, if the user has defined a function named GET-PARENT, CLISP would know not to attempt to interpret the form (GET-PARENT) as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If the user mistakenly writes (GET-PRAENT), CLISP would know he meant (GET-PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PRAENT is not the name of a variable, and that GET-PARENT is the name of a user function whose spelling is "very close" to that of GET-PRAENT. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

(1) (LIST X*FACT N), where FACT is the name of a variable, means (LIST (X*FACT) N).

(2) (LIST X*FACT N), where FACT is *not* the name of a variable but instead is the name of a function, means (LIST X*(FACT N)), i.e., N is FACT's argument.

(3) (LIST X*FACT(N)), FACT the name of a function (and not the name of a variable), means (LIST X*(FACT N)).

(4) cases (1), (2) and (3) with FACT misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics and the change would be made without the user being notified. In the other cases, the user would be informed or consulted about what was taking place. For example, to take an extreme case, suppose the expression (LIST X*FCCT N) were encountered, where there was both a function named FACT and a variable named FCT. The user would first be asked if FCCT were a misspelling of FCT. If he said YES, the expression would be interpreted as (LIST (X*FCT) N). If he said NO, the user would be asked if FCCT were a misspelling of FACT, i.e., if he intended X*FCCT N to mean X*(FACT N). If he said YES to this question, the indicated transformation would be performed. If he said NO, the system would then ask if X*FCCT should be treated as CLISP, since FCCT is not the name of a (bound) variable.[1] If he said YES, the expression would be transformed, if NO, it would be left alone, i.e., as (LIST X*FCCT N). Note that we have not even considered the case where X*FCCT is itself a misspelling of a variable name, e.g., a variable named XFCT (as with GET-PRAENT). This sort of transformation would be considered after the user said NO to X*FCCT N -> X*(FACT N).

Note: Through the discussion above, we speak of CLISP or DWIM asking the user. Actually, if the expression in question was typed in by the user for immediate execution, the user is simply informed of the transformation, on the grounds that the user would prefer an occasional misinterpretation rather than being continuously bothered, especially since he can always retype what he intended if a mistake occurs, and ask the programmer's assistant to UNDO the effects of the mistaken operations if necessary. For transformations on expressions in user programs, the user can inform CLISP whether he wishes to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) the user will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing the user.

CLISP can also handle parentheses errors caused by typing 8 or 9 for "(" or ")". (On most terminals, 8 and 9 are the lower case characters for "(" and ")", i.e., "(" and 8 appear on the same key, as do ")" and 9.) For example, if the user writes N*8FACTORIAL N-1, the parentheses error can be detected and fixed before the infix operator * is converted to the Interlisp function TIMES. CLISP is able to distinguish this situation from cases like N*8*X meaning (TIMES N 8 X), or N*8X, where 8X is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled! For example, CLISP can successfully translate the definition of FACTORIAL:

```
(IFF N=0 THENN1 ESLE N*8FACTTORIALNN-1)
```

to the corresponding COND, while making 5 spelling corrections and fixing the parenthesis error.[2]

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits the user to say things like:

```
(FOR OLD X FROM M TO N DO (PRINT X) WHILE (PRIMEP X))
```

However, the user can also write OLD (X←M), (OLD X←M), (OLD (X←M)), permute the order of the operators, e.g., (DO PRINT X TO N FOR OLD X←M WHILE PRIMEP X), omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, FROM, TO, DO, or WHILE, or leave out the word DO entirely! And, of course, he can also misspell PRINT, PRIMEP, M or N! In this example, the only thing the user could not misspell is the first X, since it specifies the *name* of the variable of iteration. The other two instances of X could be misspelled.

[1] This question is important because Interlisp users may have programs that employ identifiers containing CLISP operators. Thus, if CLISP encounters the expression A/B in a context where either A or B are not the names of variables, it will ask the user if A/B is intended to be CLISP, in case the user really does have a free variable named A/B.

[2] CLISP also contains a facility for converting from Interlisp back to CLISP, so that after running the above incorrect definition of FACTORIAL, the user could "clispify" the now correct version to obtain (IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)).

CLISP is well integrated into the Interlisp system. For example, the above iterative statement translates into an following equivalent Interlisp form using PROG, COND, GO, etc. When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation. Similarly, the compiler "knows" to compile the translated form. However, if the user PRETTYPRINTs his program, PRETTYPRINT "knows" to print the original CLISP at the corresponding point in his function. Similarly, when the user edits his program, the editor keeps the translation invisible to the user. If the user modifies the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant (page 8.1). Whereas the programmer's assistant is an invisible intermediary agent between the user's console requests and the Interlisp executive, CLISP sits between the user's programs and the Interlisp interpreter.

Only a small effort has been devoted to defining the core syntax of CLISP. Instead, most of the effort has been concentrated on providing a facility which "makes sense" out of the input expressions using context information as well as built-in and acquired information about user and system programs. It has been said that communication is based on the intention of the speaker to produce an effect in the recipient. CLISP operates under the assumption that what the user said was *intended* to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide the user with many different ways of saying the same thing, but to enable him to worry less about the *syntactic* aspects of his communication with the system. In other words, it gives the user a new degree of freedom by permitting him to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

DWIM and CLISP are invoked on iterative statements because CAR of the iterative statement is not the name of a function, and hence generates an error. If the user defines a function by the same name as an i.s. operator, e.g., WHILE, TO, etc., the operator will no longer have the CLISP interpretation when it appears as CAR of a form, although it will continue to be treated as an i.s. operator if it appears in the interior of an i.s. To alert the user, a warning message is printed, e.g., (WHILE DEFINED, THEREFORE DISABLED IN CLISP).

16.1 CLISP INTERACTION WITH USER

Syntactically and semantically well formed CLISP transformations are always performed without informing the user. Other CLISP transformations described in the previous section, e.g., misspellings of operands, infix operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (page 15.1). That is, if DWIM has been enabled in TRUSTING mode, or the transformation is in an expression typed in by the user for immediate execution, user approval is not requested, but the user is informed.[3] However, if the transformation involves a user program, and DWIM was enabled in CAUTIOUS mode, the user will be asked to approve. If he says NO, the transformation is not performed. Thus, in the previous section, phrases such as "one of these (transformations) succeeds" and "the transformation LAST-ELL -> LAST-EL would be found" etc., all mean if the user is in

[3] However, in certain situations, DWIM will ask for approval even if DWIM is enabled in TRUSTING mode. For example, the user will always be asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in LAST-ELL -> LAST-EL.

CAUTIOUS mode and the error is in a program, the corresponding transformation will be performed only if the user approves (or defaults by not responding). If the user says NO, the procedure followed is the same as though the transformation had not been found. For example, if A*B appears in the function FOO, and B is not bound (and no other transformations are found) the user would be asked A*B [IN FOO] TREAT AS CLISP ?[4]

If the user approved, A*B would be transformed to (ITIMES A B), which would then cause a U.B.A. B error in the event that the program was being run (remember the entire discussion also applies to DWIMIFYing). If the user said NO, A*B would be left alone.[5]

16.2 CLISP CHARACTER OPERATORS

CLISP recognizes a number of special characters operators, both prefix and infix, which are translated into common expressions. For example, the character + is recognized to represent addition, so CLISP translates the litatom A+B to the form (IPLUS A B). Note that CLISP is envoked, and this translation is made, only if an error occurs, such as an unbound atom error or an undefined function error for the perfectly legitamate litatom A+B. Therefore the user may choose not to use these facilities with no penalty, similar to other CLISP facilities.

The user has a lot of flexability in using CLISP character operators. A list, can always be substituted for a litatom, and vice versa, without changing the interpretation of a phrase. For example, if the value of (FOO X) is A, and the value of (FIE Y) is B, then (LIST (FOO X)+(FIE Y)) has the same value as (LIST A+B). Note that the first expression is a list of *four* elements: the atom "LIST", the list "(FOO X)", the atom "+", and the list "(FIE X)", whereas the second expression, (LIST A+B), is a list of only *two* elements: the litatom "LIST" and the litatom "A+B". Since (LIST (FOO X)+(FIE Y)) is indistinguishable from (LIST (FOO X)␣+␣(FIE Y)) because spaces before or after parentheses have no effect on the Interlisp READ program,[6] to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat (LIST A+␣B), (LIST A␣+B), and (LIST A␣+␣B) the same as (LIST A+B).

+	[CLISP Operator]
-	[CLISP Operator]
*	[CLISP Operator]
/	[CLISP Operator]
↑	[CLISP Operator]

CLISP recognizes +, -, *, /, and ↑ as the normal arithmetic infix operators. - is also recognized as the prefix operator, unary minus. These are converted to IPLUS, IDIFFERENCE (or in the case of unary minus, IMINUS), ITIMES, IQUOTIENT, and EXPT.

[4] The waiting time on such interactions is three times as long as for simple corrections, i.e., 3*DWIMWAIT.

[5] If the value of CLISPHELPFLG=NIL (initally T), the user will not be asked to approve any clisp transformation. Instead, in those situations where approval would be required, the effect is the same as though the user had been asked and said NO.

[6] CLISP does not use its own special READ program because this would require the user to explicitly identify CLISP expressions, instead of being able to intermix Interlisp and CLISP.

The I in IPLUS denotes integer arithmetic, i.e., IPLUS converts its arguments to integers, and returns an integer value. Interlisp also contains floating point arithmetic functions as well as mixed arithmetic functions. Floating point arithmetic functions are used in the translation if one or both of the operands are themselves floating point numbers, e.g., X+1.5 translates as (FPLUS X 1.5). In addition, CLISP contains a facility for declaring which type of arithmetic is to be used, either by making a global declaration, or by separate declarations about individual functions or variables (see page 16.9).

The usual precedence rules apply (although these can be easily changed by the user), i.e., * has higher precedence than + so that A+B*C is the same as A+(B*C), and both * and / are lower than ↑ so that 2*X↑2 is the same as 2*(X↑2). Operators of the same precedence group from left to right, e.g., A/B/C is equivalent to (A/B)/C. Minus is binary whenever possible, i.e., except when it is the first operator in a list, as in (-A) or (-A), or when it immediately follows another operator, as in A*-B. Note that grouping with parentheses can always be used to override the normal precedence grouping, or when the user is not sure how a particular expression will parse. The complete order of precedence for CLISP operators is given below.

Note that + in front of a number will disappear when the number is read, e.g., (FOO X +2) is indistinguishable from (FOO X 2). This means that (FOO X +2) will not be interpreted as CLISP, or be converted to (FOO (IPLUS X 2)). Similarly, (FOO X -2) will not be interpreted the same as (FOO X-2). To circumvent this, always type a space between the + or - and a number if an infix operator is intended, e.g., write (FOO X + 2).

= [CLISP Operator]
GT [CLISP Operator]
LT [CLISP Operator]
GE [CLISP Operator]
LE [CLISP Operator]

These are infix operators for "Equal", "Greater Than", "Less Than", "Greater Than or Equal", and "Less Than or Equal".

GT, LT, GE, and LE are all affected by the same declarations as + and *, with the initial default to use IGREATERP and ILESSP.

Note that only single character operators, e.g., +, ←, =, etc., can appear in the *interior* of an atom. All other operators must be set off from identifiers with spaces. For example, XLTY will not be recognized as CLISP. In some cases, DWIM will be able to diagnose this situation as a run-on spelling error, in which case after the atom is split apart, CLISP will be able to perform the indicated transformation.

A number of lisp functions, such as EQUAL, MEMBER, AND, OR, etc., can also be treated as CLISP infix operators.[7] AND is higher than OR, and both AND and OR are lower than the other infix operators, so

[7] Currently the complete list is MEMBER, MEMB, FMEMB, ILESSP, IGREATERP, LESSP, GREATERP, FGTP, EQ, NEQ, EQP, EQUAL, OR, and AND. New infix operators can be easily added, as described in page 16.21. Spelling correction on misspelled infix operators is peformed using CLISPINFIXSPLST as a spelling list.

(X OR Y AND Z) is the same as (X OR (Y AND Z)), and (X AND Y EQUAL Z) is the same as (X AND (Y EQUAL Z)). All of the infix predicates have lower precedence than Interlisp forms, since it is far more common to apply a predicate to two forms, than to use a Boolean as an argument to a function. Therefore, (FOO X GT FIE Y) is translated as ((FOO X) GT (FIE Y)), rather than as (FOO (X GT (FIE Y))). However, the user can easily change this.

: [CLISP Operator]

x:*N* extracts the *N*th element of the list *x*. FOO:3 specifies the third element of FOO, or (CADDR FOO). If *N* is less than zero, this indicates elements counting from the end of the list; i.e. FOO:-1 is the last element of FOO. : operators can be nested, so FOO:1:2 means the second element of the first element of FOO, or (CADAR FOO).

The : operator can also be used for extracting substructures of records (see page 3.1). Record operations are implemented by replacing expressions of the form X:FOO by (fetch FOO of X). Both lower and upper case are acceptable.

: is also used to indicate operations in the pattern match facility (page 23.1).

:: [CLISP Operator]

x:*N*, returns the *N*th *tail* of the list *x*. For example, FOO::3 is (CDDDR FOO), and FOO::-1 is (LAST FOO).

← [CLISP Operator]

← is used to indicate assignment. For example, X←Y translates to (SETQ X Y). If X does not have a value, and is not the name of one of the bound variables of the function in which it appears, spelling correction is attempted. However, since this may simply be a case of assigning an initial value to a new free variable, DWIM will always ask for approval before making the correction.

In conjunction with : and ::, ← can also be used to perform a more general type of assignment, involving structure modification. For example, X:2←Y means "make the second element of X *be* Y", in Interlisp terms (RPLACA (CDR X) Y). Note that the *value* of this operation is the value of RPLACA, which is (CDR X), rather than Y. Negative numbers can also be used, e.g., X:-2←Y, which translates to (RPLACA (NLEFT X 2) Y).

The user can indicate he wants /RPLACA and /RPLACD used (undoable version of RPLACA and RPLACD, see page 8.22), or FRPLACA and FRPLACD (fast versions of RPLACA and RPLACD, see page 2.15), by means of CLISP declarations (page 16.9). The initial default is to use RPLACA and RPLACD.

← is also used to indicate assignment in record operations (X:FOO←Y translates to (replace FOO of X with Y).), and pattern match operations (page 23.1).

← has different precedence on the left from on the right. On the left, ← is a "tight" operator, i.e., high precedence, so that A+B←C is the same as A+(B←C). On the right, ← has broader scope so that A←B+C is the same as A←(B+C).

On typein, $←*FORM* (<esc>←*FORM*) is equivalent to set the "last thing mentioned".[8] For example, immediately after examining the value of LONGVARIABLENAME, the user could set it by typing $← followed by a form.

Note that an atom of the form X←Y, appearing at the top level of a PROG, will *not* be recognized as an assignment statement because it will be interpreted as a PROG label by the Interlisp interpreter, and therefore will not cause an error, so DWIM and CLISP will never get to see it. Instead, one must write (X←Y).

< [CLISP Operator]
> [CLISP Operator]

Angle brackets are used in CLISP to indicate list construction. The appearance of a "<" corresponds to a "(" and indicates that a list is to be constructed containing all the elements up to the corresponding ">". For example, <A B <C>> translates to (LIST A B (LIST C)). ! can be used to indicate that the next expression is to be inserted in the list as a *segment*, e.g., <A B ! C> translates to (CONS A (CONS B C)) and <! A ! B C> to (APPEND A B (LIST C)). !! is used to indicate that the next expression is to be inserted as a segment, and furthermore, all list structure to its right in the angle brackets is to be physically attached to it, e.g., <!! A B> translates to (NCONC1 A B), and <!!A !B !C> to (NCONC A (APPEND B C)). Not (NCONC (APPEND A B) C), which would have the same value, but would attach C to B, and not attach either to A. Note that <, !, !!, and > need not be separate atoms, for example, <A B ! C> may be written equally well as < A B !C >. Also, arbitrary Interlisp or CLISP forms may be used within angle brackets. For example, one can write <FOO←(FIE X) ! Y> which translates to (CONS (SETQ FOO (FIE X)) Y). CLISPIFY converts expressions in CONS, LIST, APPEND, NCONC, NCONC1, /NCONC, and /NCONC1 into equivalent CLISP expressions using <, >, !, and !!.

Note: brackets differ from other CLISP operators. For example, <A B 'C> translates to (LIST A B (QUOTE C)) even though following ', all *operators* are ignored for the rest of the identifier. (This is true only if a previous unmatched < has been seen, e.g., (PRINT 'A>B) will print the atom A>B.) Note however that <A B '␣C> D> is equivalent to (LIST A B (QUOTE C>) D).

' [CLISP Operator]

CLISP recognizes ' as a prefix operator. ' means QUOTE when it is the first character in an identifier, and is ignored when it is used in the interior of an identifier. Thus, X='Y means (EQ X (QUOTE Y)), but X=CAN'T means (EQ X CAN'T), *not* (EQ X CAN) followed by (QUOTE T). This enables users to have variable and function names with ' in them (so long as the ' is not the first character).

Following ', all operators are ignored for the rest of the identifier, e.g., '*A means (QUOTE *A), and 'X=Y means (QUOTE X=Y), not (EQ (QUOTE X) Y). To write (EQ (QUOTE X) Y), one writes Y='X, or 'X =Y. This is one place where an extra space does make a difference.

[8] i.e., is equivalent to (SET LASTWORD *FORM*). See page 15.15.

On typein, '$ (i.e., '<esc>) is equivalent to (QUOTE *VALUE-OF-LASTWORD*) (see page 15.15). For example, after calling PRETTYPRINT on LONGFUNCTION, the user could move its definition to FOO by typing (MOVD '$ 'FOO).[9]

~ [CLISP Operator]

CLISP recognizes ~ as a prefix operator meaning NOT. ~ can negate a form, as in ~(ASSOC X Y), or ~X, or negate an infix operator, e.g., (A ~GT B) is the same as (A LEQ B). Note that ~A=B means (EQ (NOT A) B).

When ~ negates an operator, e.g., ~=, ~LT, the two operators are treated as a single operator whose precedence is that of the second operator. When ~ negates a function, e.g., (~FOO X Y), it negates the whole form, i.e., (~(FOO X Y)).

Order of Predeence of CLISP Operators:
'
:
← (left precedence)
- (unary), ~
↑
*, /
+, - (binary)
← (right precedence)
=
Interlisp forms
LT, GT, EQUAL, MEMBER, etc.
AND
OR
IF, THEN, ELSEIF, ELSE
iterative statement operators

16.3 DECLARATIONS

CLISP declarations are used to affect the choice of Interlisp function used as the translation of a particular operator. For example, A+B can be translated as either (IPLUS A B), (FPLUS A B), or (PLUS A B), depending on the declaration in effect. Similarly X:1←Y can mean (RPLACA X Y), (FRPLACA X Y), or (/RPLACA X Y), and <!!A B> either (NCONC1 A B) or (/NCONC1 A B). Note that the choice of function on all CLISP transformations are affected by the CLISP declaration in effect, i.e., iterative statements, pattern matches, record operations, as well as infix and prefix operators.

(CLISPDEC *DECLST*) [Function]

Puts into effect the declarations in *DECLST*. CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

[9] *Not* (MOVD $ 'FOO), which would be equivalent to (MOVD LONGFUNCTION 'FOO), and would (probably) cause a U.B.A. LONGFUNCTION error, nor MOVD($ FOO), which would actually move the definition of $ to FOO, since DWIM and the spelling corrector would never be invoked.

The user can makes (changes) a global declaration by calling CLISPDEC with *DECLST* a list of declarations, e.g., (CLISPDEC '(FLOATING UNDOABLE)). Changing a global declaration does not affect the speed of subsequent CLISP transformations, since all CLISP transformation are table driven (i.e., property list), and global declarations are accomplished by making the appropriate internal changes to CLISP at the time of the declaration. If a function employs *local* declarations (described below), there will be a slight loss in efficiency owing to the fact that for each CLISP transformation, the declaration list must be searched for possibly relevant declarations.

Declarations are implemented in the order that they are given, so that later declarations override earlier ones. For example, the declaration FAST specifies that FRPLACA, FRPLACD, FMEMB, and FLAST be used in place of RPLACA, RPLACD, MEMB, and LAST; the declaration RPLACA specifies that RPLACA be used. Therefore, the declarations (FAST RPLACA RPLACD) will cause FMEMB, FLAST, RPLACA, and RPLACD to be used.

The initial global declaration is INTEGER and STANDARD.

The table below gives the declarations available in CLISP, and the Interlisp functions they indicate:

Declaration	Interlisp Functions to be used
INTEGER or FIXED	IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP
FLOATING	FPLUS, FMINUS, FDIFFERENCE, FTIMES, FQUOTIENT, LESSP, FGREATERP
MIXED	PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP
FAST	FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC
UNDOABLE	/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON
STANDARD	RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON
RPLACA, RPLACD, /RPLACA, etc.	corresponding function

16.3.1 Local Declarations

The user can also make local declarations affecting a selected function or functions by inserting an expression of the form (CLISP: . *DECLARATIONS*) immediately following the argument list, i.e., as CADDR of the definition. Such local declarations take precedence over global declarations. Declarations affecting selected variables can be indicated by lists, where the first element is the name of a variable, and the rest of the list the declarations for that variable. For example, (CLISP: FLOATING (X INTEGER)) specifies that in this function integer arithmetic be used for computations involving X, and floating arithmetic for all other computations.[10] The user can also make local record declarations by inserting a record declaration, e.g., (RECORD --), (ARRAYRECORD --), etc., in the local declaration list. In addition, a local declaration of the form (RECORDS A B C) is equivalent to having copies of the global declarations A, B, and C in the local declaration. Local record declarations override global record declarations for the function in which they appear. Local declarations can also be used to override the global setting of certain DWIM/CLISP parameters effective only for transformations within that function, by including in the local declaration an expression of the form (*VARIABLE* = *VALUE*), e.g., (PATVARDEFAULT = QUOTE).

The CLISP: expression is converted to a comment of a special form recognized by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed in a function, this comment will be searched for a relevant declaration, and if one is found, the corresponding function will be used. Otherwise, if none are found, the global declaration(s) currently in effect will be used.

Local declarations are effective in the order that they are given, so that later declarations can be used to override earlier ones, e.g., (CLISP: FAST RPLACA RPLACD) specifies that FMEMB, FLAST, RPLACA, and RPLACD be used. An exception to this is that declarations for specific variables take precedence of general, function-wide declarations, regardless of the order of appearance, as in (CLISP: (X INTEGER) FLOATING).

CLISPIFY also checks the declarations in effect before selecting an infix operator to ensure that the corresponding CLISP construct would in fact translate back to this form. For example, if a FLOATING declaration is in effect, CLISPIFY will convert (FPLUS X Y) to X+Y, but leave (IPLUS X Y) as is. Note that if (FPLUS X Y) is CLISPIFYed while a FLOATING declaration is under effect, and then the declaration is changed to INTEGER, when X+Y is translated back to Interlisp, it will become (IPLUS X Y).

16.4 CLISP OPERATION

CLISP is a part of the basic Interlisp system. Without any special preparations, the user can include CLISP constructs in programs, or type them in directly for evaluation (in EVAL or APPLY format), then, when the "error" occurrs, and DWIM is called, it will destructively transform the CLISP to the equivalent Interlisp expression and evaluate the Interlisp expression. CLISP transformations, like all DWIM corrections, are undoable. User approval is not requested, and no message is printed.[11]

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if the user writes (LIST X+Y*), the error diagnostic MISSING OPERAND AT X+Y* IN (LIST X+Y*) would be generated. Similarly, if the user writes (LAST+EL X), CLISP knows that ((IPLUS LAST EL) X) is not a valid Interlisp expression, so the error diagnostic MISSING OPERATOR IN (LAST+EL X) is generated. (For example, the user might have meant to

[10] "involving" means where the variable itself is an operand. For example, with the declaration (FLOATING (X INTEGER)) in effect, (FOO X)+(FIE X) would translate to FPLUS, i.e., use floating arithmetic, even though X appears somewhere inside of the operands, whereas X+(FIE X) would translate to IPLUS. If there are declarations involving *both* operands, e.g., X+Y, with (X FLOATING) (Y INTEGER), whichever appears first in the declaration list will be used.

[11] This entire discussion also applies to CLISP transformation initiated by calls to DWIM from DWIMIFY.

say(LAST+EL*X).) Note that if LAST+EL were the name of a defined function, CLISP would never see this form.

Since the bad CLISP transformation might not be CLISP at all, for example, it might be a misspelling of a user function or variable, DWIM holds all CLISP error messages until after trying other corrections. If one of these succeeds, the CLISP message is discarded. Otherwise, if all fail, the message is printed (but no change is made).[12] For example, suppose the user types (R/PLACA X Y). CLISP generates a diagnostic, since ((IQUOTIENT R PLACA) X Y) is obviously not right. However, since R/PLACA spelling corrects to /RPLACA, this diagnostic is never printed.

If a CLISP infix construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound,[13] it is possible that either the operand is misspelled, e.g., the user wrote X+YY for X+Y, or that a CLISP transformation operation was not intended at all, but that the entire expression is a misspelling. For example, if the user has a variable named LAST-EL, and writes (LIST LAST-ELL). Therefore, CLISP computes, but does not actually perform, the indicated infix transformation. DWIM then continues, and if it is able to make another correction, does so, and ignores the CLISP interpretation. For example, with LAST-ELL, the transformation LAST-ELL -> LAST-EL would be found.

If no other transformation is found, and DWIM is about to interpret a construct as CLISP for which one of the operands is not bound, DWIM will ask the user whether CLISP was intended, in this case by printing LAST-ELL TREAT AS CLISP ?[14]

The same sort of procedure is followed with 8 and 9 errors. For example, suppose the user writes FOO8*X where FOO8 is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks the user to approve FOO8*X -> FOO (*X. (For example, this would make sense if the user has (or plans to define) a function named *X.) If he refuses, the user is asked whether FOO8*X is to be treated as CLISP. Similarly, if FOO8 were the name of a variable, and the user writes FOOO8*X, he will first be asked to approve FOOO8*X -> FOOO (XX,[15] and if he refuses, then be offered the FOOO8 -> FOO8 correction.

CLISP also contains provision for correcting misspellings of infix operators (other than single characters), IF words, and i.s. operators. This is implemented in such a way that the user who does not misspell them is not penalized. For example, if the user writes IF N=0 THEN 1 ELSSE N*(FACT N-1) CLISP does *not* operate by checking each word to see if it is a misspelling of IF, THEN, ELSE, or ELSEIF, since this would seriously degrade CLISP's performance on *all* IF statements. Instead, CLISP assumes that all of the IF words are spelled correctly, and transforms the expression to (COND ((ZEROP N) 1 ELSSE N*(FACT N-1))). Later, after DWIM cannot find any other interpretation for ELSSE, and using the

[12] Except that CLISP error messages are not printed on type-in. For example, typing X+*Y will just produce a U.B.A. X+*Y message.

[13] For the purpose of DWIMIFYing, "not bound" means no top level value, not on list of bound variables built up by DWIMIFY during its analysis of the expression, and not on NOFIXVARSLST, i.e., not previously seen.

[14] If more than one infix operator was involved in the CLISP construct, e.g., X+Y+Z, or the operation was an assignment to a variable already noticed, or TREATASCLISPFLG is T (initially NIL), the user will simply be informed of the correction, e.g., X+Y+Z TREATED AS CLISP. Otherwise, even if DWIM was enabled in TRUSTING mode, the user will be asked to approve the correction.

[15] The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or undefined function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

fact that this atom originally appeared in an IF statement, DWIM attempts spelling correction, using (IF THEN ELSE ELSEIF) for a spelling list. When this is successful, DWIM "fails" all the way back to the original IF statement, changes ELSSE to ELSE, and starts over. Misspellings of AND, OR, LT, GT, etc. are handled similarly.

CLISP also contains many Do-What-I-Mean features besides spelling corrections. For example, the form (LIST +X Y) would generate a MISSING OPERATOR error. However, (LIST -X Y) makes sense, if the minus is unary, so DWIM offers this interpretation to the user. Another common error, especially for new users, is to write (LIST X*FOO(Y)) or (LIST X*FOO Y), where FOO is the name of a function, instead of (LIST X*(FOO Y)). Therefore, whenever an operand that is not bound is also the name of a function (or corrects to one), the above interpretations are offered.

16.5 CLISP TRANSLATIONS

The translation of CLISP character operators and the CLISP word IF are handled by *replacing* the CLISP expression with the corresponding Interlisp expression, and discarding the original CLISP.[16] This is done because (1) the CLISP expression is easily recomputable (by CLISPIFY) and (2) the Interlisp expressions are simple and straightforward. Another reason for discarding the original CLISP is that it may contain errors that were corrected in the course of translation (e.g., FOO←FOOO:1, N*8FOO X), etc.). If the original CLISP were retained, either the user would have to go back and fix these errors by hand, thereby negating the advantage of having DWIM perform these corrections, or else DWIM would have to keep correcting these errors over and over.

Note that CLISPIFY is sufficiently fast that it is practical for the user to configure his Interlisp system so that all expressions are automatically CLISPIFYed immediately before they are presented to him. For example, he can define an edit macro to use in place of P which calls CLISPIFY on the current expression before printing it. Similarly, he can inform PRETTYPRINT to call CLISPIFY on each expression before printing it, etc.

Where (1) or (2) are not the case, e.g., with iterative statements, pattern matches, record expressions, etc. the original CLISP *is* retained (or a slightly modified version thereof), and the translation is stored[17] elsewhere, usually in the hash array CLISPARRAY.[18] The interpreter automatically checks this array when

[16] If CLISPIFTRANFLG is T, the original CLISP for IF statements (modified to correct errors) is retained. See page 16.20.

[17] by the function CLISPTRAN (page 16.19).

[18] The user can also indicate that he wants the original CLISP retained by embedding it in an expression of the form (CLISP . *CLISP-EXPRESSION*), e.g., (CLISP X:5:3) or (CLISP <A B C ! D>). In such cases, the translation will be stored remotely as described in the text. Furthermore, such expressions will be treated as CLISP even if infix and prefix transformations have been disabled by setting CLISPFLG to NIL (page 16.19). In other words, the user can instruct the system to interpret as CLISP infix or prefix constructs only those expressions that are specifically flagged as such. The user can also include CLISP declarations by writing (CLISP *DECLARATIONS* . *FORM*), e.g., (CLISP (CLISP: FLOATING) ...). These declarations will be used in place of any CLISP declarations in the function definition. Note this feature provides a way of including CLISP declarations in macro definitions.

given a form CAR of which is not a function.[19] Similarly, the compiler performs a GETHASH when given a form it does not recognize to see if it has a translation, which is then compiled instead of the form. Whenever the user *changes* a CLISP expresson by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or dwimified, the expression will be retranslated.[20] The function PPT and the edit commands PPT and CLISP: are available for examining translations (page 16.20). If PRETTYTRANFLG is T, PRETTYPRINT will print the translations instead of the corresponding CLISP expression (see page 16.20). This can be used for exporting programs to systems that do not provide CLISP, and to examine translations for debugging purposes.

16.6 DWIMIFY

DWIMIFY is effectively a preprocessor for CLISP. DWIMIFY operates by scanning an expression as though it were being interpreted, and for each form that would generate an error, calling DWIM to "fix" it. DWIMIFY performs *all* DWIM transformations, not just CLISP transformations, so it does spelling correction, fixes 8-9 errors, handles F/L, etc. Thus the user will see the same messages, and be asked for approval in the same situations, as he would if the expression were actually run. If DWIM is unable to make a correction, no message is printed, the form is left as it was, and the analysis proceeds.

DWIMIFY knows exactly how the interpreter works. It knows the syntax of PROGs, SELECTQs, LAMBDA expressions, SETQs, et al. It knows that the argument of NLAMBDAs are not evaluated.[21] It also knows how variables are bound.[22] In the course of its analysis of a particular expression, DWIMIFY builds a list of the bound variables from the LAMBDA expressions and PROGs that it encounters. It uses this list for spelling corrections. DWIMIFY also knows not to try to "correct" variables that are on this list since they would be bound if the expression were actually being run. However, note that DWIMIFY cannot, a priori, know about variables that are used freely but would be bound in a higher function if the expression were evaluated in its normal context. Therefore, DWIMIFY will try to "correct" these variables.[23] Similarly, DWIMIFY will attempt to correct forms for which CAR is undefined, even when the form is not in error from the user's standpoint, but the corresponding function has simply not yet been defined.

[19] CLISP translations can also be used to supply an interpretation for function objects, as well as forms, either for function objects that are used openly, i.e., appearing as CAR of form, function objects that are explicitly APPLYed, as with arguments to mapping functions, or function objects contained in function definition cells. In all cases, if CAR of the object is not LAMBDA or NLAMBDA, the interpreter and compiler will check CLISPARRAY.

[20] If the value of CLISPRETRANFLG is T, DWIMIFY will also (re)translate any expressions which have translations stored remotely. See page 16.16.

[21] The user can inform DWIMIFY that an NLAMBDA function *does* evaluate its arguments (presumably by direct calls to EVAL), by including on its property list the property INFO with value EVAL or a list which contains the atom EVAL.

[22] The user can inform DWIMIFY that a particular function or construct binds variables by including the atom BINDS on the INFO property for CAR of the form. In this case, DWIMIFY assumes that CADR of the form is the variable list, i.e. a list of atoms, or lists of the form (*VAL VALUE*). LAMBDA, NLAMBDA, PROG, and RESETVARS are handled in this fashion.

[23] DWIMIFY rebinds FIXSPELLDEFAULT to N, so that if the user is not at the terminal when dwimifying (or compiling), spelling corrections will not be performed.

DWIMIFY will also inform the user when it encounters an expression with too *many* arguments,[24] because such an occurrence, although does not cause an error in the Interlisp interpreter, nevertheless is frgquently symptomatic of a parenthesis error. For example, if the user wrote (CONS (QUOTE FOO X)) instead of (CONS (QUOTE FOO) X), DWIMIFY will print:

```
POSSIBLE PARENTHESIS ERROR IN
(QUOTE FOO X)
TOO MANY ARGUMENTS (MORE THAN 1)
```

DWIMIFY will also check to see if a PROG label contains a clisp character,[25] and if so, will alert the user by printing the message SUSPICIOUS PROG LABEL, followed by the label. The PROG label will *not* be treated as CLISP.

Note that in most cases, an attempt to transform a form that is already as the user intended will have no effect (because there will be nothing to which that form could reasonably be transformed). However, in order to avoid needless calls to DWIM or to avoid possible confusion, the user can inform DWIMIFY *not* to attempt corrections or transformations on certain functions or variables by adding them to the list NOFIXFNSLST or NOFIXVARSLST respectively. Note that the user could achieve the same effect by simply setting the corresponding variables, and giving the functions dummy definitions.

DWIMIFY will never attempt corrections on global variables, i.e., variables that are a member of the list GLOBALVARS, or have the property GLOBALVAR with value T, on their property list. Similarly, DWIMIFY will not attempt to correct variables declared to be SPECVARS in block declarations or via DECLARE expressions in the function body. The user can also declare variables that are simply used freely in a function by using the USEDFREE declaration.

DWIMIFY and DWIMIFYFNS (used to DWIMIFY several functions) maintain two internal lists of those functions and variables for which corrections were unsuccessfully attempted. These lists are initialized to the values of NOFIXFNSLST and NOFIXVARSLST. Once an attempt is made to fix a particular function or variable, and the attempt fails, the function or variable is added to the corresponding list, so that on subsequent occurrences (within this call to DWIMIFY or DWIMIFYFNS), no attempt at correction is made. For example, if FOO calls FIE several times, and FIE is undefined at the time FOO is dwimified, DWIMIFY will not bother with FIE after the first occurrence. In other words, once DWIMIFY "notices" a function or variable, it no longer attempts to correct it. DWIMIFY and DWIMIFYFNS also "notice" free variables that are set in the expression being processed. Moreover, once DWIMIFY "notices" such functions or variables, it subsequently treats them the same as though they were actually defined or set.

Note that these internal lists are local to each call to DWIMIFY and DWIMIFYFNS, so that if a function containing FOOO, a misspelled call to FOO, is DWIMIFYed before FOO is defined or mentioned, if the function is DWIMIFYed again after FOO has been defined, the correction will be made.

The user can undo selected transformations performed by DWIMIFY, as described on page 8.11.

(DWIMIFY *X* *QUIETFLG* *L*) [Function]

Performs all DWIM and CLISP corrections and transformations on *X* that would be performed if *X* were run, and prints the result unless *QUIETFLG*=T.

[24] unless DWIMCHECK#ARGSFLG=NIL (initially T).

[25] unless DWIMCHECKPROGLABELSFLG=NIL (initially T), or the label is a member of NOFIXVARSLST.

If *x* is an atom and *L* is NIL, *x* is treated as the name of a function, and its entire definition is dwimified. If *x* is a list or *L* is not NIL, *x* is the expression to be dwimified. If *L* is not NIL, it is the edit push-down list leading to *x*, and is used for determining context, i.e., what bound variables would be in effect when *x* was evaluated, whether *x* is a form or sequence of forms, e.g., a COND clause, etc.

If *x* is an iterative statement and *L* is NIL, DWIMIFY will also print the translation, i.e., what is stored in the hash array.

(DWIMIFYFNS *FN*$_1$ ··· *FN*$_N$) [NLambda NoSpread Function]

Dwimifies each of the functions given. If only one argument is given, it is evalued. If its value is a list, the functions on this list are dwimified. If only one argument is given, it is atomic, its value is not a list, and it is the name of a known file, DWIMIFYFNS will operate on (FILEFNSLST *FN*$_1$), e.g. (DWIMIFYFNS FOO.LSP) will dwimify every function in the file FOO.LSP.

Every 30 seconds, DWIMIFYFNS prints the name of the function it is processing, a la PRETTYPRINT.

Value is a list of the functions dwimified.

NOFIXFNSLST [Variable]

List of functions that DWIMIFY will not try to correct.

NOFIXVARSLST [Variable]

List of variables that DWIMIFY will not try to correct.

NOSPELLFLG [Variable]

If T, DWIMIFY will not perform any spelling corrections. Initially NIL. NOSPELLFLG is reset to T when compiling functions whose definitions are obtained from a file, as opposed to being in core.

CLISPHELPFLG [Variable]

If NIL, DWIMIFY will not ask the user for approval of any CLISP transformations. Instead, in those situations where approval would be required, the effect is the same as though the user had been asked and said NO. Initially T.

DWIMIFYCOMPFLG [Variable]

If T, DWIMIFY is called before compiling an expression. Initially NIL.

DWIMCHECK#ARGSFLG [Variable]

If T, causes DWIMIFY to check for too many arguments in a form. Initially T.

DWIMCHECKPROGLABELSFLG [Variable]

If T, causes DWIMIFY to check whether a PROG label contains a CLISP character. Initially T.

DWIMESSGAG [Variable]

If T, suppresses all DWIMIFY error messages. Initially NIL.

CLISPRETRANFLG [Variable]

If T, informs DWIMIFY to (re)translate all expressions which have remote translations in the CLISP hash array. Initially NIL.

16.7 CLISPIFY

CLISPIFY converts Interlisp expressions to CLISP. Note that the expression given to CLISPIFY need *not* have originally been input as CLISP, i.e., CLISPIFY can be used on functions that were written before CLISP was even implemented. CLISPIFY is cognizant of declaration rules as well as all of the precedence rules. For example, CLISPIFY will convert (IPLUS A (ITIMES B C)) into A+B*C, but (ITIMES A (IPLUS B C)) into A*(B+C). CLISPIFY handles such cases by first DWIMIFYing the expression. CLISPIFY also knows how to handle expressions consisting of a mixture of Interlisp and CLISP, e.g., (IPLUS A B*C) is converted to A+B*C, but (ITIMES A B+C) to (A*(B+C)). CLISPIFY converts calls to the six basic mapping functions, MAP, MAPC, MAPCAR, MAPLIST, MAPCONC, and MAPCON, into equivalent iterative statements. It also converts certain easily recognizable internal PROG loops to the corresponding iterative statements. CLISPIFY can convert all iterative statements input in CLISP back to CLISP, regardless of how complicated the translation was, because the original CLISP is saved.

CLISPIFY is not destructive to the original Interlisp expression, i.e., CLISPIFY produces a new expression without changing the original.[26] CLISPIFY will not convert expressions appearing as arguments to NLAMBDA functions.[27]

Note: Disabling a CLISP operator with CLDISABLE (page 16.19) will also disable the corresponding CLISPIFY transformation. Thus, if ← is "turned off", A←B will not transform to (SETQ A B), nor vice versa.

(CLISPIFY *X* *L*) [Function]

Clispifies *X*. If *X* is an atom and *L* is NIL, *X* is treated as the name of a function, and its definition (or EXPR property) is clispified. After CLISPIFY has finished, *X* is redefined (using /PUTD) with its new CLISP definition. The value of CLISPIFY is *X*. If *X* is atomic and not the name of a function, spelling correction is attempted. If this fails, an error is generated.

If *X* is a list, or *L* is not NIL, *X* itself is the expression to be clispified. If *L* is not NIL, it is the edit push-down list leading to *X* and is used to determine context as with DWIMIFY, as well as to obtain the local declarations, if any. The value of CLISPIFY is the clispified version of *X*.

(CLISPIFYFNS FN_1 ··· FN_N) [NLambda NoSpread Function]

Like DWIMIFYFNS (page 16.16) except calls CLISPIFY instead of DWIMIFY.

[26] The new expression may however contain some "pieces" of the original, since CLISPIFY attempts to minimize the number of CONSes by not copying structure whenever possible.

[27] Except for those functions whose INFO property is or contains the atom EVAL. CLISPIFY also contains built in information enabling it to process special forms such as PROG, SELECTQ, etc. If the INFO property is or contains the atom LABELS, CLISPIFY will never create an atom (by packing) at the top level of the expression. PROG is handled in this fashion.

CL:FLG [Variable]

Affects CLISPIFY's handling of forms beginning with CAR, CDR, ··· CDDDDR, as well as pattern match and record expressions. If CL:FLG is NIL, these are not transformed into the equivalent : expressions. This will prevent CLISPIFY from constructing any expression employing a : infix operator, e.g., (CADR X) will not be transformed to X:2. If CL:FLG is T, CLISPIFY will convert to : notation only when the argument is atomic or a simple list (a function name and one atomic argument). If CL:FLG is ALL, CLISPIFY will convert to : expressions whenever possible.

CL:FLG is initially T.

CLREMPARSFLG [Variable]

If T, CLISPIFY will remove parentheses in certain cases from simple forms, where "simple" means a function name and one or two atomic arguments. For example, (COND ((ATOM X) --)) will CLISPIFY to (IF ATOM X THEN --). However, if CLREMPARSFLG is set to NIL, CLISPIFY will produce (IF (ATOM X) THEN --). Note that regardless of the setting of this flag, the expression can be input in either form.

CLREMPARSFLG is initially NIL.

CLISPIFYPACKFLG [Variable]

CLISPIFYPACKFLG affects the treatment of infix operators with atomic operands. If CLISPIFYPACKFLG is T, CLISPIFY will pack these into single atoms, e.g., (IPLUS A (ITIMES B C)) becomes A+B*C. If CLISPIFYPACKFLG is NIL, no packing is done, e.g., the above becomes A⊔+⊔B⊔*⊔C.

CLISPIFYPACKFLG is initially T.

CLISPIFYUSERFN [Variable]

If T, causes the function CLISPIFYUSERFN, which should be a function of one argument, to be called on each form (list) not otherwise recognized by CLISPIFY. If a non-NIL value is returned, it is treated as the clispified form. Initially NIL

Note that CLISPIFYUSERFN must be both set and defined to use this feature.

FUNNYATOMLST [Variable]

Suppose the user has variables named A, B; and A*B. If CLISPIFY were to convert (ITIMES A B) to A*B, A*B would not translate back correctly to (ITIMES A B), since it would be the name of a variable, and therefore would not cause an error. The user can prevent this from happening by adding A*B to the list FUNNYATOMLST. Then, (ITIMES A B) would CLISPIFY to A⊔*⊔B.

Note that A*B's appearance on FUNNYATOMLST would *not* enable DWIM and CLISP to decode A*B+C as (IPLUS A*B C); FUNNYATOMLST is used only by CLISPIFY. Thus, if an identifier contains a CLISP character, it should always be separated (with spaces) from other operators. For example, if X* is a variable, the user should write (SETQ X* *FORM*) in CLISP as X*⊔←*FORM*, not X*←*FORM*. In general, it is best to avoid use of identifiers containing CLISP character operators as much as possible.

16.8 MISCELLANEOUS FUNCTIONS AND VARIABLES

CLISPFLG [Variable]

If set to NIL, disables all CLISP infix or prefix transformations (but does not affect IF/THEN/ELSE statements, or iterative statements).

If CLISPFLG=TYPE-IN, CLISP transformations are performed only on expressions that are typed in for evaluation, i.e., not on user programs.

If CLISPFLG=T, CLISP transformations are performed on all expressions.

The initial value for CLISPFLG is T. CLISPIFYing anything will cause CLISPFLG to be set to T.

CLISPCHARS [Variable]

A list of the operators that can appear in the interior of an atom. Currently (+ - * / ↑ ~ ' = ← : < > +- ~= @ !).

CLISPCHARRAY [Variable]

A bit table of the characters on CLISPCHARS used for calls to STRPOSL (page 2.31). CLISPCHARRAY is initialized by performing (SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS)).

CLISPINFIXSPLST [Variable]

A list of infix operators used for spelling correction.

CLISPARRAY [Variable]

Hash array used for storing CLISP translations. CLISPARRAY is checked by FAULTEVAL and FAULTAPPLY on erroneous forms before calling DWIM, and by the compiler.

(CLISPTRAN *X* *TRAN*) [Function]

Gives *X* the translation *TRAN* by storing (key *X*, value *TRAN*) in the hash array CLISPARRAY. CLISPTRAN is called for all CLISP translations, via a non-linked, external function call, so it can be advised.

(CLISPDEC *DECLST*) [Function]

Puts into effect the declarations in *DECLST* (see page 16.9). CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

(CLDISABLE *OP*) [Function]

Disables the CLISP operator *OP*. For example, (CLDISABLE '-) makes - be just another character. CLDISABLE can be used on all CLISP operators, e.g., infix operators, prefix operators, iterative statement operators, etc. CLDISABLE is undoable.

Note: Simply removing a character operator from CLISPCHARS will prevent it from being treated as a CLISP operator when it appears as part of an atom, but it will continue to be an operator when it appears as a separate atom, e.g. (FOO + X) vs FOO+X.

CLISPIFTRANFLG [Variable]

Affects handling of translations of IF|THEN|ELSE statements (see page 4.4). If T, the translations are stored elsewhere, and the (modified) CLISP retained. If NIL, the corresponding COND expression replaces the CLISP. Initially T.

CLISPIFYPRETTYFLG [Variable]

If non-NIL, causes PRETTYPRINT (and therefore PP and MAKEFILE) to CLISPIFY selected function definitions before printing them according to the following interpretations of CLISPIFYPRETTYFLG:

ALL	Clispify all functions.
T or EXPRS	Clispify all functions currently defined as EXPRs.
CHANGES	Clispify all functions marked as having been changed.
a list	Clispify all functions in that list.

CLISPIFYPRETTYFLG is (temporarily) reset to T when MAKEFILE is called with the option CLISPIFY, and reset to CHANGES when the file being dumped has the property FILETYPE value CLISP. CLISPIFYPRETTYFLG is initially NIL.

Note: If CLISPIFYPRETTYFLG is non-NIL, and the only transformation performed by DWIM are well formed CLISP transformations, i.e., no spelling corrections, the function will *not* be marked as changed, since it would only have to be re-clispified and re-prettyprinted when the file was written out.

PRETTYTRANFLG [Variable]

If T, causes PRETTYPRINT to print translations instead of CLISP expressions. This is useful for exporting to a LISP system that does not have CLISP. PRETTYTRANFLG is (temporarily) reset to T when MAKEFILE is called with the option NOCLISP. PRETTYTRANFLG is initially NIL.

(PPT *x*) [NLambda NoSpread Function]

Both a function and an edit macro for prettyprinting translations. It performs a PP after first resetting PRETTYTRANFLG to T, thereby causing any translations to be printed instead of the corresponding CLISP.

CLISP: [Editor Command]

Edit macro that obtains the translation of the correct expression, if any, from CLISPARRAY, and calls EDITE on it.

CL [Editor Command]

Edit macro. Replaces current expression with CLISPIFYed current expression. Current expression can be an element or tail.

DW [Editor Command]

Edit macro. DWIMIFYs current expression, which can be an element (atom or list) or tail.

Both CL and DW can be called when the current expression is either an element or a tail and will work properly. Both consult the declarations in the function being edited, if any, and both are undoable.

(LOWERCASE *FLG*) [Function]

If *FLG*=T, LOWERCASE makes the necessary internal modifications so that CLISPIFY will use lower case versions of AND, OR, IF, THEN, ELSE, ELSEIF, and all i.s. operators. This produces more readable output. Note that the user can always type in *either* upper or lower case (or a combination), regardless of the action of LOWERCASE. If *FLG*=NIL, CLISPIFY will use uppercase versions of AND, OR, et al. The value of LOWERCASE is its previous "setting". LOWERCASE is undoable. The initial setting for LOWERCASE is T.

16.9 CLISP INTERNAL CONVENTIONS

CLISP is almost entirely table driven by the property lists of the corresponding infix or prefix operators. For example, much of the information used for translating the + infix operator is stored on the property list of the litatom "+". Thus it is relatively easy to add new infix or prefix operators or change old ones, simply by adding or changing selected property values. (There *is* some built in information for handling minus, :, ', and ~, i.e., the user could not himself add such "special" operators, although he can disable or redefine them.)

Global declarations operate by changing the LISPFN and CLISPINFIX properties of the appropriate operators.

CLISPTYPE [Property Name]

The property value of the property CLISPTYPE is the precedence number of the operator: higher values have higher precedence, i.e., are tighter. Note that the actual value is unimportant, only the value relative to other operators. For example, CLISPTYPE for :, ↑, and * are 14, 6, and 4 respectively. Operators with the same precedence group left to right, e.g., / also has precedence 4, so A/B*C is (A/B)*C.

An operator can have a different left and right precedence by making the value of CLISPTYPE be a dotted pair of two numbers, e.g., CLISPTYPE of ← is (8 . -12). In this case, CAR is the left precedence, and CDR the right, i.e., CAR is used when comparing with operators on the *left*, and CDR with operators on the *right*. For example, A*B←C+D is parsed as A*(B←(C+D)) because the left precedence of ← is 8, which is higher than that of *, which is 4. The right precedence of ← is -12, which is lower than that of +, which is 2.

If the CLISPTYPE property for any operator is removed, the corresponding CLISP transformation is disabled, as well as the inverse CLISPIFY transformation.

UNARYOP [Property Name]

The value of property UNARYOP must be T for unary operators or brackets. The operand is always on the right, i.e., unary operators or brackets are always prefix operators.

BROADSCOPE [Property Name]

The value of property BROADSCOPE is T if the operator has lower precedence than Interlisp forms, e.g., LT, EQUAL, AND, etc. For example, (FOO X AND Y) parses as ((FOO X) AND Y). If the BROADSCOPE property were removed from the property list of AND, (FOO X AND Y) would parse as (FOO (X AND Y)).

LISPFN [Property Name]

The value of the property LISPFN is the name of the function to which the infix operator translates. For example, the value of LISPFN for ↑ is EXPT, for ' QUOTE, etc. If the value of the property LISPFN is NIL, the infix operator itself is also the function, e.g., AND, OR, EQUAL.

SETFN [Property Name]

If FOO has a SETFN property FIE, then (FOO --)←X translates to (FIE -- X). For example, if the user makes ELT be an infix operator, e.g. #, by putting appropriate CLISPTYPE and LISPFN properties on the property list of # then he can also make # followed by ← translate to SETA, e.g., X#N←Y to (SETA X N Y), by putting SETA on the property list of ELT under the property SETFN. Putting the list (ELT) on the property list of SETA under property SETFN will enable SETA forms to CLISPIFY back to ELT's.

CLISPINFIX [Property Name]

The value of this property is the CLISP infix to be used in CLISPIFYing. This property is stored on the property list of the corresponding Interlisp function, e.g., the value of property CLISPINFIX for EXPT is ↑, for QUOTE is ' etc.

CLISPWORD [Property Name]

Appears on the property list of clisp operators which can appear as CAR of a form, such as FETCH, REPLACE, IF, iterative statement operators, etc. Value of property is of the form (*KEYWORD* . *NAME*), where *NAME* is the lowercase version of the operator, and *KEYWORD* is its type, e.g. FORWORD, IFWORD, RECORDWORD, etc.

KEYWORD can also be the name of a function. When the atom appears as CAR of a form, the function is applied to the form and the result taken as the correct form. In this case, the function should either physically change the form, or call CLISPTRAN (page 16.19) to store the translation.

As an example, to make & be an infix character operator meaning OR, the user could do the following:

```
←(PUTPROP '& 'CLISPTYPE (GETPROP 'OR 'CLISPTYPE))
←(PUTPROP '& 'LISPFN 'OR)
←(PUTPROP '& 'BROADSCOPE T)
←(PUTPROP 'OR 'CLISPINFIX '&)
←(SETQ CLISPCHARS (CONS '& CLISPCHARS))
←(SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS))
```

CHAPTER 17

THE TELETYPE EDITOR

The Interlisp teletype editor allows rapid, convenient modification of list structures. Most often it is used to edit function definitions, (often while the function itself is running) via the function EDITF, e.g., EDITF(FOO). However, the editor can also be used to edit the value of a variable, via EDITV, to edit a property list, via EDITP, or to edit an arbitrary expression, via EDITE. It is an important feature which allows good on-line interaction in the Interlisp system.

In Interlisp-D, most editing is done using the display editor DEdit (page 20.1), which is-an extended, display-oriented version of the teletype editor. The teletype editor is still available, as it offers a facility for doing complex modifications of program structure under program control. For example, BREAKIN (page 10.5) calls the teletype editor to insert a function break within the body of a function. By calling the function EDITMODE (page 20.2) it is possible to set the "default editor" (TELETYPE or DISPLAY) called by Masterscope, the break package, etc.

This chapter begins with a lengthy introduction intended for the new user. The reference portion begins on page 17.9.

17.1 INTRODUCTION

Let us introduce some of the basic editor commands, and give a flavor for the editor's language structure by guiding the reader through a hypothetical editing session. Suppose we are editing the following incorrect definition of APPEND:

```
[LAMBDA (X)
    Y
    (COND
       ((NUL X)
        Z)
       (T (CONS (CAR)
                (APPEND (CDR X Y]
```

We call the editor via the function EDITF:

```
←EDITF(APPEND)
EDIT
*
```

The editor responds by typing EDIT followed by *, which is the editor's prompt character. This signifies that the editor is ready to accept commands. In the examples in this chapter, all lines beginning with * were typed by the user, the rest by the editor.

At any given moment, the editor's attention is centered on some substructure of the expression being edited. This substructure is called the *current expression*, and it is what the user sees when he gives the editor the command P, for print. Initially, the current expression is the top level one, i.e., the entire expression being edited. Thus:

```
*P
(LAMBDA (X) Y (COND & &))
*
```

Note that the editor prints the current expression as though printlevel (page 6.18) were set to (2 . 20), i.e., sublists of sublists are printed as &, tails of long lists printed as --. The command ? will print the current expression as though printlevel were 1000.

```
*?
(LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR) (APPEND (CDR X Y))))))
*
```

and the command PP will prettyprint the current expression.

A positive integer is interpreted by the editor as a command to descend into the correspondingly numbered element of the current expression. Thus:

```
*2
*P
(X)
*
```

A negative integer has a similar effect, but counting begins from the end of the current expression and proceeds backward, i.e., -1 refers to the last element in the current expression, -2 the next to the last, etc. For either positive integer or negative integer, if there is no such element, an error occurs. "Editor errors" are not the same as Interlisp function errors, i.e., they never cause breaks or even go through the error machinery but are direct calls to ERROR! indicating that a command is in some way faulty. What happens next depends on the context in which the command was being executed. For example, there are conditional commands which branch on errors. In most situations, though, an error will cause the editor to type the faulty command followed by a ? and wait for more input. Note that typing control-E while a command is being executed aborts the command exactly as though it had caused an error. *The current expression is never changed when a command causes an error.* Thus:

```
*P
(X)
*2

2  ?
*1
*P
X
*
```

A phrase of the form "the current expression is changed" or "the current expression becomes" refers to a shift in the editor's attention, not to a modification of the structure being edited.

When the user changes the current expression by descending into it, the old current expression is not lost. Instead, the editor actually operates by maintaining a *chain* of expressions leading to the current one. The current expression is simply the last link in the chain. Descending adds the indicated subexpression onto the end of the chain, thereby making it be the current expression. The command 0 is used to ascend the chain; it removes the last link of the chain, thereby making the *previous* link be the current expression. Thus:

```
*P
X
*0 P
(X)
*0 -1 P
(COND (& Z) (T &))
*
```

Note the use of several commands on a single line in the previous output. The editor operates in a line buffered mode, the same as EVALQT. Thus no command is actually seen by the editor, or executed, until the line is terminated, either by a carriage return, or a matching right parenthesis. The user can thus use control-A and control-Q for line-editing edit commands, the same as he does for inputs to the Interlisp executive.

In our editing session, we will make the following corrections to APPEND: delete Y from where it appears, add Y to the end of the argument list, change NUL to NULL, change Z to Y, add X after CAR, and insert a right parenthesis following CDR X.

First we will delete Y. By now we have forgotten where we are in the function definition, but we want to be at the "top" so we use the command ↑, which ascends through the entire chain of expressions to the top level expression, which then becomes the current expression, i.e., ↑ removes all links except the first one.

```
*↑ P
(LAMBDA (X) Y (COND & &))
*
```

Note that if we are already at the top, ↑ has no effect, i.e., it is a no-op. However, 0 would generate an error. In other words, ↑ means "go to the top," while 0 means "ascend one link."

The basic structure modification commands in the editor are:

(*N*) ($N \geq 1$) [Editor Command]

Deletes the corresponding element from the current expression.

(*N* $E_1 \cdots E_M$) ($N \geq 1$) [Editor Command]

Replaces the *N*th element in the current expression with $E_1 \cdots E_M$.

(-*N* $E_1 \cdots E_M$) ($N \geq 1$) [Editor Command]

Inserts $E_1 \cdots E_M$ before the *N*th element in the current expression.

Thus:

```
*P
(LAMBDA (X) Y (COND & &))
*(3)
*(2 (X Y))
*P
```

```
(LAMBDA (X Y) (COND & &))
*
```

All structure modification done by the editor is destructive, i.e., the editor uses RPLACA *and* RPLACD *to physically change the structure it was given.*

Note that all three of the above commands perform their operation with respect to the *N*th element from the front of the current expression; the sign of *N* is used to specify whether the operation is replacement or insertion. Thus, there is no way to specify deletion or replacement of the *N*th element from the end of the current expression, or insertion before the *N*th element from the end without counting out that element's position from the front of the list. Similarly, because we cannot specify insertion after a particular element, we cannot attach something at the end of the current expression using the above commands. Instead, we use the command N (for NCONC). Thus we could have performed the above changes instead by:

```
*P
(LAMBDA (X) Y (COND & &))
*(3)
*2 (N Y)
*P
(X Y)
*↑ P
*(LAMBDA (X Y) (COND & &))
*
```

Now we are ready to change NUL to NULL. Rather than specify the sequence of descent commands necessary to reach NUL, and then replace it with NULL, e.g., 3 2 1 (1 NULL), we will use F, the find command, to find NUL:

```
*P
(LAMBDA (X Y) (COND & &))
*F NUL
*P
(NUL X)
*(1 NULL)
*0 P
((NULL X) Z)
*
```

Note that F is special in that it corresponds to *two* inputs. In other words, F says to the editor, "treat your *next* command as an expression to be searched for." The search is carried out in printout order in the current expression. If the target expression is not found there, F automatically ascends and searches those portions of the higher expressions that would appear after (in a printout) the current expression. If the search is successful, the new current expression will be the structure where the expression was found,[1] and the chain will be the same as one resulting from the appropriate sequence of ascent and descent

[1] If the search is for an atom, e.g., F NUL, the current expression will be the structure containing the atom.

commands. If the search is not successful, an error occurs, and neither the current expression nor the chain is changed:[2]

```
*P
((NULL X) Z)
*F COND P

COND ?
*P
*((NULL X) Z)
*
```

Here the search failed to find a COND following the current expression, although of course a COND does appear earlier in the structure. This last example illustrates another facet of the error recovery mechanism: to avoid further confusion when an error occurs, all commands on the line *beyond* the one which caused the error (and all commands that may have been typed ahead while the editor was computing) are forgotten.

We could also have used the R command (for Replace) to change NUL to NULL. A command of the form (R E_1 E_2) will replace all occurrences of E_1 in the current expression by E_2. There must be at least one such occurrence or the R command will generate an error. Let us use the R command to change all Z's (even though there is only one) in APPEND to Y:

```
*↑ (R Z Y)
*F Z

Z ?
*PP
[LAMBDA (X Y)
   (COND
      ((NULL X)
        Y)
      (T (CONS (CAR)
               (APPEND (CDR X Y]
*
```

The next task is to change (CAR) to (CAR X). We could do this by (R (CAR) (CAR X)), or by:

```
*F CAR
*(N X)
*P
(CAR X)
*
```

The expression we now want to change is the next expression after the current expression, i.e., we are currently looking at (CAR X) in (CONS (CAR X) (APPEND (CDR X Y))). We could get to the

[2] F is never a no-op, i.e., if successful, the current expression after the search will never be the same as the current expression before the search. Thus F *EXPR* repeated without intervening commands that change the edit chain can be used to find successive instances of *EXPR*.

APPEND expression by typing 0 and then 3 or -1, or we can use the command NX, which does both operations:

```
*P
(CAR X)
*NX P
(APPEND (CDR X Y))
*
```

Finally, to change (APPEND (CDR X Y)) to (APPEND (CDR X) Y), we could perform (2 (CDR X) Y), or (2 (CDR X)) and (N Y), or 2 and (3), deleting the Y, and then 0 (N Y). However, if Y were a complex expression, we would not want to have to retype it. Instead, we could use a command which effectively inserts and/or removes left and right parentheses. There are six of these commands: BI ("Both In"), BO ("Both Out"), LI ("Left In"), LO ("Left Out"), RI ("Right In"), and RO ("Right Out"). Of course, we will always have the same number of left parentheses as right parentheses, because the parentheses are just a notational guide to structure that is provided by our print program. Herein lies one of the principal advantages of a LISP oriented editor over a text editor: unbalanced parentheses errors are not possible. Thus, LI, LO, RI, and RO actually do not insert or remove just one parenthesis, but this is very suggestive of what actually happens.

In this case, we would like a right parenthesis to appear following X in (CDR X Y). Therefore, we use the command (RI 2 2), which means insert a right parentheses after the second element in the second element (of the current expression):

```
*P
(APPEND (CDR X Y))
*(RI 2 2)
*P
(APPEND (CDR X) Y)
*
```

We have now finished our editing, and can exit from the editor, to test APPEND, or we could test it while still inside of the editor, by using the E command:

```
*E APPEND((A B) (C D E))
(A B C D E)
*
```

The E command causes the next input to be evaluated by Interlisp. If there is another input following it, as in the above example, the first will be applied (with APPLY) to the second. Otherwise, the input is evaluated (with EVAL).

We prettyprint APPEND, and leave the editor.

```
*PP
[LAMBDA (X Y)
   (COND
      ((NULL X)
        Y)
      (T (CONS (CAR X)
               (APPEND (CDR X) Y]
*OK
```

```
APPEND
←
```

17.2 COMMANDS FOR THE NEW USER

As mentioned earlier, the Interlisp manual is intended primarily as a reference manual, and the remainder of this chapter is organized and presented accordingly. While the commands introduced in the previous scenario constitute a complete set, i.e., the user could perform any and all editing operations using just those commands, there are many situations in which knowing the right command(s) can save the user considerable effort. We include here as part of the introduction a list of those commands which are not only frequently applicable but also easy to use. They are not presented in any particular order, and are all discussed in detail in the reference portion of the chapter.

UNDO [Editor Command]

Undoes the last modification to the structure being edited, e.g., if the user deletes the wrong element, UNDO will restore it. The availability of UNDO should give the user confidence to experiment with any and all editing commands, no matter how complex, because he can always reverse the effect of the command.

BK [Editor Command]

Like NX, except makes the expression immediately *before* the current expression become current.

BF [Editor Command]

Backwards Find. Like F, except searches backwards, i.e., in inverse print order.

\ [Editor Command]

Restores the current expression to the expression before the last "big jump", e.g., a find command, an ↑, or another \. For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR.

\P [Editor Command]

Like \ except it restores the edit chain to its state as of the last print, either by P, ?, or PP. If the edit chain has not been changed since the last print, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

Thus if the user types P followed by 3 2 1 P, \P will take him back to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take him back to the second P. Thus the user can use \P to flip back and forth between two current expressions.

The search expression given to the F or BF command need not be a literal expression. Instead, it can be a pattern. The symbol & can be used anywhere within this pattern to match with any single *element* of a list, and -- can be used to match with any *segment* of a list. Thus, in the incorrect definition of APPEND used earlier, F (NUL &) could have been used to find (NUL X), and F (CDR --) or F (CDR & &), but not F (CDR &), to find (CDR X Y).

Note that & and -- can be nested arbitrarily deeply in the pattern. For example, if there are many places where the variable X is set, F SETQ may not find the desired expression, nor may F (SETQ X &). It may be necessary to use F (SETQ X (LIST --)). However, the usual technique in such a case is to pick out a unique atom which occurs prior to the desired expression, and perform two F commands. This "homing in" process seems to be more convenient than ultra-precise specification of the pattern.

$ (<esc>) is equivalent to -- at the character level, e.g., VER$ will match with VERYLONGATOM, as will $ATOM, $LONG$, (but not $LONG) and VNM. $ can be nested inside of a pattern, e.g., F (SETQ VER$ (CONS --)).

If the search is successful, the editor will print = followed by the atom which matched with the $-atom, e.g.,

```
*F (SETQ VER$ &)
=VERYLONGATOM
*
```

Frequently the user will want to replace the entire current expression, or insert something before it. In order to do this using a command of the form (N E_1 ··· E_M) or (-N E_1 ··· E_M), the user must be *above* the current expression. In other words, he would have to perform a 0 followed by a command with the appropriate number. However, if he has reached the current expression via an F command, he may not know what that number is. In this case, the user would like a command whose effect would be to modify the edit chain so that the current expression became the first element in a new, higher current expression. Then he could perform the desired operation via (1 E_1 ··· E_M) or (-1 E_1 ··· E_M). UP is provided for this purpose.

UP [Editor Command]

After UP operates, the old current expression is the first element of the new current expression. Note that if the current expression happens to be the first element in the next higher expression, then UP is exactly the same as 0. Otherwise, UP modifies the edit chain so that the new current expression is a proper tail (page 2.19) of the next higher expression:

```
*F APPEND P
(APPEND (CDR X) Y)
*UP P
··· (APPEND & Y))
*0 P
(CONS (CAR X) (APPEND & Y))
*
```

The ··· is used by the editor to indicate that the current expression is a *tail* of the next higher expression as opposed to being an element (i.e., a member) of the next higher expression. Note: if the current expression is *already* a tail, UP has no effect.

(B E_1 ··· E_M) [Editor Command]

Inserts E_1 ··· E_M before the current expression, i.e., does an UP and then a (-1 E_1 ··· E_M).

(A E_1 ··· E_M) [Editor Command]

Inserts E_1 ··· E_M after the current expression, i.e., does an UP and then either a (-2 E_1 ··· E_M) or an (N E_1 ··· E_M), if the current expression is the last one in the next higher expression.

(: E_1 ··· E_M) [Editor Command]

Replaces the current expression by E_1 ··· E_M, i.e., does an UP and then a (1 E_1 ··· E_M).

DELETE [Editor Command]

Deletes the current expression; equivalent to (:).

Earlier, we introduced the RI command in the APPEND example. The rest of the commands in this family: BI, BO, LI, LO, and RO, perform similar functions and are useful in certain situations. In addition, the commands MBD and XTR can be used to combine the effects of several commands of the BI-BO family. MBD (page 17.28) is used to embed the current expression in a larger expression. For example, if the current expression is (PRINT *bigexpression*), and the user wants to replace it by (COND (FLG (PRINT *bigexpression*))), he could accomplish this by (LI 1), (-1 FLG), (LI 1), and (-1 COND), or by a single MBD command.

XTR (page 17.27) is used to eXTRact an expression from the current expression. For example, extracting the PRINT expression from the above COND could be accomplished by (1), (LO 1), (1), and (LO 1) or by a single XTR command. The new user is encouraged to include XTR and MBD in his repertoire as soon as he is familiar with the more basic commands.

17.3 LOCAL ATTENTION-CHANGING COMMANDS

This section describes commands that change the current expression (i.e., change the edit chain) thereby "shifting the editor's attention." These commands depend only on the *structure* of the edit chain, as compared to the search commands (presented later), which search the contents of the structure.

UP [Editor Command]

UP modifies the edit chain so that the old current expression (i.e., the one at the time UP was called) is the first element in the new current expression. If the current expression is the first element in the next higher expression UP simply does a 0. Otherwise UP adds the corresponding tail to the edit chain.

If a P command would cause the editor to type ··· before typing the current expression, ie., the current expression is a tail of the next higher expression, UP has no effect.

For Example:

```
*PP
(COND ((NULL X) (RETURN Y)))
*1 P
COND
*UP P
(COND (& &))
*-1 P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y))
*UP P
```

```
... ((NULL X) (RETURN Y)))
*F NULL P
(NULL X)
*UP P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
```

The execution of UP is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is (A NIL B NIL C NIL) and the user performs 4 followed by UP, the current expression should then be ... NIL C NIL). UP can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, LASTAIL. Thus after the 4 command is executed, LASTAIL is (NIL C NIL). When UP is called, it first determines if the current expression is a tail of the next higher expression. If it is, UP is finished. Otherwise, UP computes (MEMB *CURRENT-EXPRESSION NEXT-HIGHER-EXPRESSION*) to obtain a tail beginning with the current expression.[3] If there are no other instances of the current expression in the next higher expression, this tail is the correct one. Otherwise UP uses LASTAIL to select the correct tail.[4]

N (*N*≥1) [Editor Command]

Adds the *N*th element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least *N* elements.

-*N* (*N*≥1) [Editor Command]

Adds the *N*th element from the end of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least *N* elements.

0 [Editor Command]

Sets the edit chain to CDR of the edit chain, thereby making the next higher expression be the new current expression. Generates an error if there is no higher expression, i.e., CDR of edit chain is NIL.

Note that 0 usually corresponds to going back to the next higher left parenthesis, but not always. For example:

[3] The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example the user has directly (and incorrectly) manipulated the edit chain, UP generates an error.

[4] Occasionally the user can get the edit chain into a state where LASTAIL cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were EQ, and the user descended more than one level into one of them and then tried to come back out using UP. In this case, UP prints LOCATION UNCERTAIN and generates an error. Of course, we could have solved this problem completely in our implementation by saving at each descent *both* elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The LASTAIL solution is cheap and resolves 99% of the ambiguities.

```
*P
(A B C D E F B)
*3 UP P
... C D E F G)
*3 UP P
... E F G)
*0 P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command !0 can be used.

!0 [Editor Command]

Does repeated 0's until it reaches a point where the current expression is *not* a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.

↑ [Editor Command]

Sets the edit chain to LAST of edit chain, thereby making the top level expression be the current expression. Never generates an error.

NX [Editor Command]

Effectively does an UP followed by a 2, thereby making the current expression be the next expression. Generates an error if the current expression is the last one in a list. (However, !NX described below will handle this case.)

BK [Editor Command]

Makes the current expression be the previous expression in the next higher expression. Generates an error if the current expression is the first expression in a list.

For example,

```
*PP
(COND ((NULL X) (RETURN Y)))
*F RETURN P
(RETURN Y)
*BK P
(NULL X)
```

Both NX and BK operate by performing a !0 followed by an appropriate number, i.e., there won't be an extra tail above the new current expression, as there would be if NX operated by performing an UP followed by a 2.

(NX *N*) [Editor Command]

($N \geq 1$) Equivalent to *N* NX commands, except if an error occurs, the edit chain is not changed.

(BK *N*) [Editor Command]

($N \geq 1$) Equivalent to *N* BK commands, except if an error occurs, the edit chain is not changed.

Note: (NX -*N*) is equivalent to (BK *N*), and vice versa.

!NX [Editor Command]

Makes the current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression. For example:

```
*PP
(PROG ((L L)
       (UF L))
  LP  (COND
         ((NULL (SETQ L (CDR L)))
            (ERROR!))
         ([NULL (CDR (FMEMB (CAR L) (CADR L]
            (GO LP)))
      (EDITCOM (QUOTE NX))
      (SETQ UNFIND UF)
      (RETURN L))
*F CDR P
(CDR L)
*NX

NX   ?
*!NX P
(ERROR!)
*!NX P
((NULL &) (GO LP))
*!NX P
(EDITCOM (QUOTE NX))
*
```

!NX operates by doing 0's until it reaches a stage where the current expression is *not* the last expression in the next higher expression, and then does a NX. Thus !NX always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e., !NX and NX always produce different results. For example using the previous current expression:

```
*F CAR P
(CAR L)
*!NX P
(GO LP)
*\P P
(CAR L)
*NX P
(CADR L)
*
```

(NTH *N*) [Editor Command]

(*N* ≠ 0) Equivalent to *N* followed by UP, i.e., causes the list starting with the *N*th element of the current expression (or *N*th from the end if *N* < 0) to become the current expression. Causes an error if current expression does not have at least *N* elements.

(NTH 1) is a no-op, as is (NTH -*L*) where *L* is the length of the current expression.

line-feed [Editor Command]

Moves to the "next" expression and prints it, i.e. performs a NX if possible, otherwise performs a !NX. (The latter case is indcated by first printing ">".)

control-X [Editor Command]

Control-X[5] moves to the "previous" thing and then prints it, i.e. performs a BK if possible, otherwise a !0 followed by a BK.

control-Z [Editor Command]

Control-Z[6] moves to the last expression and prints it, i.e. does -1 followed by P.

Line-feed, control-X, and control-Z are implemented as *immediate* read macros; as soon as they are read, they abort the current printout. They thus provide a convenient way of moving around in the editor. In order to facilitate using different control characters for those macros, the function SETTERMCHARS is provided (see page 17.59).

17.4 COMMANDS THAT SEARCH

All of the editor commands that search use the same pattern matching routine (the function EDIT4E, page 17.57). We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern *PAT* matches with *X* if any of the following conditions are true:

(1) If *PAT* is EQ to *X*.

(2) If *PAT* is &.

(3) If *PAT* is a number and EQP to *X*.

(4) If *PAT* is a string and (STREQUAL *PAT* *X*) is true.

(5) If (CAR *PAT*) is the atom *ANY*, (CDR *PAT*) is a list of patterns, and one of the patterns on (CDR *PAT*) matches *X*.

(6) If *PAT* is a literal atom or string containing one or more $s (<esc>s), each $ can match an indefinite number (including 0) of contiguous characters in the atom or string *X*, e.g., VER$ matches both VERYLONGATOM and "VERYLONGSTRING" as do $LONG$ (but not $LONG), and VLT. Note: the atom $ (<esc>) matches only with itself.

(7) If *PAT* is a literal atom or string ending in *two* <esc>s, *PAT* matches with the atom or string *X* if it is "close" to *PAT*, in the sense used by the spelling corrector (page 15.13). E.g. CONSS$$ matches with CONS, CNONC$$ with NCONC or NCONC1.

[5] Control-A in Interlisp on TOPS-20.

[6] Control-L in Interlisp on TOPS-20.

The pattern matching routine always types a message of the form =*MATCHING-ITEM* to inform the user of the object matched by a pattern of the above two types, unless EDITQUIETFLG=T. For example, if VER$ matches VERYLONGATOM, the editor would print =VERYLONGATOM.

(8) If (CAR *PAT*) is the atom --, *PAT* matches *x* if (CDR *PAT*) matches with some tail of *x*. For example, (A -- (&)) will match with (A B C (D)), but not (A B C D), or (A B C (D) E). However, note that (A -- (&) --) will match with (A B C (D) E). In other words, -- can match any interior segment of a list.

If (CDR *PAT*)= NIL, i.e., *PAT*=(--), then it matches any tail of a list. Therefore, (A --) matches (A), (A B C) and (A . B).

(9) If (CAR *PAT*) is the atom ==, *PAT* matches *x* if and only if (CDR *PAT*) is EQ to *x*.

This pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command *typed* in by the user obviously cannot be EQ to already existing structure.

(10) If (CADR *PAT*) is the atom .. (two periods), *PAT* matches *x* if (CAR *PAT*) matches (CAR *x*) and (CDDR *PAT*) is contained in *x*, as described on page 17.20.

(11) Otherwise if *x* is a list, *PAT* matches *x* if (CAR *PAT*) matches (CAR *x*), and (CDR *PAT*) matches (CDR *x*).

When the editor is searching, the pattern matching routine is called to match with *elements* in the structure, unless the pattern begins with ... (three periods), in which case CDR of the pattern is matched against proper tails in the structure. Thus,

```
*P
(A B C (B C))
*F (B --)
*P
(B C)
*0 F (... B --)
*P
... B C (B C))
```

Matching is also attempted with atomic tails (except for NIL). Thus,

```
*P
(A (B . C))
*F C
*P
.... C)
```

Although the current expression is the atom C after the final command, it is printed as .... C) to alert the user to the fact that C is a *tail*, not an element. Note that the pattern C will match with either instance of C in (A C (B . C)), whereas (.... C) will match only the second C. The pattern NIL will only match with NIL as an element, i.e., it will not match in (A B), even though CDDR of (A B) is NIL. However, (.... NIL) (or equivalently (...)) may be used to specify a NIL *tail*, e.g., (...

. NIL) will match with CDR of the third subexpression of ((A . B) (C . D) (E)).

17.4.1 Search Algorithm

Searching begins with the current expression and proceeds in print order. Searching usually means find the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged. At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with ... (three periods) in which case it is matched against the next tail of the expression.

If the match is not successful, the search operation is recursive first in the CAR direction, and then in the CDR direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level. Note: A find command of the form (F *PATTERN* NIL) will only attempts matches at the top level of the current expression, i.e., it does not descend into elements, or ascend to higher expressions.

However, at no point is the total recursive depth of the search (sum of number of CARs and CDRs descended into) allowed to exceed the value of the variable MAXLEVEL. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the element or tail for which the recursive depth is below MAXLEVEL. This feature is designed to enable the user to search circular list structures (by setting MAXLEVEL small), as well as protecting him from accidentally encountering a circular list structure in the course of normal editing. MAXLEVEL can also be set to NIL, which is equivalent to infinity. MAXLEVEL is initially set to 300.

If a successful match is not found in the current expression, the search automatically ascends to the next higher expression, and continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails (or is aborted by control-E), the edit chain is not changed (nor are any CONSes performed).

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had the user reached that expression via a sequence of integer commands.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom, unless the atom is a tail, e.g., B in (A . B). In this case, the current expression will be B, but will print as ... . B). In other words, the search effectively does an UP.[7]

17.4.2 Search Commands

All of the commands below set LASTAIL for use by UP, set UNFIND for use by \ (page 17.21), and do not change the edit chain or perform any CONSes if they are unsuccessful or aborted.

F *PATTERN* [Editor Command]

Actually two commands: the F informs the editor that the *next* command is to be

[7] Unless UPFINDFLG=NIL (initially set to T). For discussion, see "Form Oriented Editing", page 17.26.

interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., `F` *PATTERN* means find the next instance of *PATTERN*.

If `(MEMB` *PATTERN CURRENT-EXPRESSION*`)` is true, `F` does not proceed with a full recursive search. If the value of the `MEMB` is `NIL`, `F` invokes the search algorithm described on page 17.15.

Note that if the current expression is `(PROG NIL LP (COND (-- (GO LP1))) ··· LP1 ···)`, then `F LP1` will find the `PROG` label, not the `LP1` inside of the `GO` expression, even though the latter appears first (in print order) in the current expression. Note that typing `1` (making the atom `PROG` be the current expression) followed by `F LP1` *would* find the first `LP1`.

`F` *PATTERN* `N` [Editor Command]

Same as `F` *PATTERN*, i.e., `F`inds the `N`ext instance of *PATTERN*, except that the `MEMB` check of `F` *PATTERN* is not performed.

`F` *PATTERN* `T` [Editor Command]

Similar to `F` *PATTERN*, except that it may succeed without changing the edit chain, and it does not perform the `MEMB` check.

For example, if the current expression is `(COND ···)`, `F COND` will look for the next `COND`, but `(F COND T)` will "stay here".

`(F` *PATTERN N*`)` [Editor Command]

($N \geq 1$) Finds the *N*th place that *PATTERN* matches. Equivalent to `(F` *PATTERN* `T)` followed by `(F` *PATTERN* `N)` repeated *N*-1 times. Each time *PATTERN* successfully matches, *N* is decremented by 1, and the search continues, until *N* reaches 0. Note that *PATTERN* does not have to match with *N* identical expressions; it just has to match *N* times. Thus if the current expression is `(FOO1 FOO2 FOO3)`, `(F FOO$ 3)` will find `FOO3`.

If *PATTERN* does not match successfully *N* times, an error is generated and the edit chain is unchanged (even if *PATTERN* matched *N*-1 times).

`(F` *PATTERN*`)` [Editor Command]
`F` *PATTERN* `NIL` [Editor Command]

Similar to `F` *PATTERN*, except that it only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing the edit chain.

For example, if the current expression is `(PROG NIL (SETQ X (COND & &)) (COND &) ...)`, the command `F COND` will find the `COND` inside the `SETQ`, whereas `(F (COND --))` will find the top level `COND`, i.e., the second one.

`(FS` $PATTERN_1$ ··· $PATTERN_N$`)` [Editor Command]

Equivalent to `F` $PATTERN_1$ followed by `F` $PATTERN_2$ ··· followed by `F` $PATTERN_N$, so that if `F` $PATTERN_M$ fails, the edit chain is left at the place $PATTERN_{M-1}$ matched.

`(F= ` *EXPRESSION X*`)` [Editor Command]

Equivalent to `(F (== .` *EXPRESSION*`)` *X*`)`, i.e., searches for a structure `EQ` to *EXPRESSION* (see page 17.13).

`(ORF` *PATTERN*$_1$ ··· *PATTERN*$_N$`)` [Editor Command]

Equivalent to `(F (*ANY*`*PATTERN*$_1$ ··· .*PATTERN*$_N$`) N)`, i.e., searches for an expression that is matched by either *PATTERN*$_1$, *PATTERN*$_2$, ··· or *PATTERN*$_N$ (see page 17.13).

`BF` *PATTERN* [Editor Command]

"Backwards Find". Searches in reverse print order, beginning with the expression immediately before the current expression (unless the current expression is the top level expression, in which case `BF` searches the entire expression, in reverse order).

`BF` uses the same pattern match routine as `F`, and `MAXLEVEL` and `UPFINDFLG` have the same effect, but the searching begins at the *end* of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point `BF` ascends and backs up, etc.

For example, if the current expression is

```
(PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --),
```

the command `F LIST` followed by `BF SETQ` will leave the current expression as `(SETQ Y (LIST Z))`, as will `F COND` followed by `BF SETQ`.

`BF` *PATTERN* `T` [Editor Command]

Similar to `BF` *PATTERN*, except that the search always includes the current expression, i.e., starts at the end of current expression and works backward, then ascends and backs up, etc.

Thus in the previous example, where `F COND` followed by `BF SETQ` found `(SETQ Y (LIST Z))`, `F COND` followed by `(BF SETQ T)` would find the `(SETQ W --)` expression.

`(BF` *PATTERN*`)` [Editor Command]
`BF` *PATTERN* `NIL` [Editor Command]

Same as `BF` *PATTERN*.

`(GO` *LABEL*`)` [Editor Command]

Makes the current expression be the first thing after the `PROG` label *LABEL*, i.e. goes where an executed `GO` would go.

17.4.3 Location Specification

Many of the more sophisticated commands described later in this chapter use a more general method of specifying position called a location specification. A location specification is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the editor are interpreted as though they had been preceded by `F`; normally such commands would cause errors. For example, the location specification `(COND 2 3)` specifies the 3rd element in the first clause of the next COND.[8]

Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a state where it detects that it is "looping", at which point it gives up. Thus, if (COND 2 3) is being located, and the first clause of the next COND contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next COND. However, if a point were reached where there were no further CONDs, then the first command, COND, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

The IF command (page 17.46) in conjunction with the ## function (page 17.46) provide a way of using arbitrary predicates applied to elements in the current expression. IF and ## will be described in detail later in the chapter, along with examples illustrating their use in location specifications.

Throughout this chapter, the meta-symbol @ is used to denote a location specification. Thus @ is a list of commands interpreted as described above. @ can also be atomic, in which case it is interpreted as (LIST @).

(LC . @) [Editor Command]

Provides a way of explicitly invoking the location operation, e.g., (LC COND 2 3) will perform the the search described above.

(LCL . @) [Editor Command]

Same as LC except the search is confined to the current expression, i.e., the edit chain is rebound during the search so that it looks as though the editor were called on just the current expression. For example, to find a COND containing a RETURN, one might use the location specification (COND (LCL RETURN) \) where the \ would reverse the effects of the LCL command, and make the final current expression be the COND.

(2ND . @) [Editor Command]

Same as (LC . @) followed by another (LC . @) except that if the first succeeds and second fails, no change is made to the edit chain.

(3ND . @) [Editor Command]

Similar to 2ND.

(← *PATTERN*) [Editor Command]

Ascends the edit chain looking for a link which matches *PATTERN*. In other words, it keeps doing 0's until it gets to a specified point. If *PATTERN* is atomic, it is matched with the first element of each link, otherwise with the entire link. If no match is found, an error is generated, and the edit chain is unchanged.

Note: If *PATTERN* is of the form (IF *EXPRESSION*), *EXPRESSION* is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues. See page 17.46.

[8] Note that the user could always write F COND followed by 2 and 3 for (COND 2 3) if he were not sure whether or not COND was the name of an atomic command.

For example:

```
*PP
[PROG NIL
   (COND
      [(NULL (SETQ L (CDR L)))
        (COND
           (FLG (RETURN L]
      ([NULL (CDR (FMEMB (CAR L)
        (CADR L]]
*F CADR
*(← COND)
*P
(COND (& &) (& &))
*
```

Note that this command differs from BF in that it does not search *inside* of each link, it simply ascends. Thus in the above example, F CADR followed by BF COND would find (COND (FLG (RETURN L))), not the higher COND.

(BELOW *COM* *X*) [Editor Command]

Ascends the edit chain looking for a link specified by *COM*, and stops *X* links below that (only links that are elements are counted, not tails). In other words BELOW keeps doing 0's until it gets to a specified point, and then backs off *X* 0's.

Note that *X* is evaluated, so one can type (BELOW *COM* (IPLUS X Y)).

(BELOW *COM*) [Editor Command]

Same as (BELOW *COM* 1).

For example, (BELOW COND) will cause the COND *clause* containing the current expression to become the new current expression. Thus if the current expression is as shown above, F CADR followed by (BELOW COND) will make the new expression be ([NULL (CDR (FMEMB (CAR L) (CADR L] (GO LP)), and is therefore equivalent to 0 0 0 0.

The BELOW command is useful for locating a substructure by specifying something it contains. For example, suppose the user is editing a list of lists, and wants to find a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

(NEX *COM*) [Editor Command]

Same as (BELOW *COM*) followed by NX.

For example, if the user is deep inside of a SELECTQ clause, he can advance to the next clause with (NEX SELECTQ).

NEX [Editor Command]

Same as (NEX ←).

The atomic form of NEX is useful if the user will be performing repeated executions of (NEX *COM*). By simply MARKing (see page 17.21) the chain corresponding to *COM*, he can use NEX to step through the sublists.

(NTH *COM*) [Editor Command]

Generalized NTH command. Effectively performs (LCL . *COM*), followed by (BELOW \), followed by UP.

If the search is unsuccessful, NTH generates an error and the edit chain is not changed.

Note that (NTH *NUMBER*) is just a special case of (NTH *COM*), and in fact, no special check is made for *COM* a number; both commands are executed identically.

In other words, NTH locates *COM*, using a search restricted to the current expression, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation. For example:

```
*P
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND UF) (RETURN L))
*(NTH UF)
*P
... (SETQ UNFIND UF) (RETURN L))
*
```

PATTERN .. @ [Editor Command]

E.g., (COND .. RETURN). Finds a COND that contains a RETURN, at any depth. Equivalent to (but more efficient than) (F *PATTERN* N), (LCL . @) followed by (← *PATTERN*).

An infix command, ".." is not a meta-symbol, it *is* the name of the command. @ is CDDR of the command. Note that (*PATTERN* .. @) can also be used directly as an edit pattern as described on page 17.13, e.g. F (*PATTERN* .. @).

For example, if the current expression is

```
(PROG NIL [COND ((NULL L) (COND (FLG (RETURN L] --),
```

then (COND .. RETURN) will make (COND (FLG (RETURN L))) be the current expression. Note that it is the innermost COND that is found, because this is the first COND encountered when ascending from the RETURN. In other words, (*PATTERN* .. @) is not *always* equivalent to (F *PATTERN* N), followed by (LCL . @) followed by \.

Note that @ is a location specification, not just a pattern. Thus (RETURN .. COND 2 3) can be used to find the RETURN which contains a COND whose first clause contains (at least) three elements. Note also that since @ permits any edit command, the user can write commands of the form (COND .. (RETURN .. COND)), which will locate the first COND that contains a RETURN that contains a COND.

17.5 COMMANDS THAT SAVE AND RESTORE THE EDIT CHAIN

Several facilities are available for saving the current edit chain and later retrieving it: MARK, which marks the current chain for future reference, ←, which returns to the last mark without destroying it, and ←←, which returns to the last mark and also erases it.

MARK [Editor Command]

Adds the current edit chain to the front of the list MARKLST.

← [Editor Command]

Makes the new edit chain be (CAR MARKLST). Generates an error if MARKLST is NIL, i.e., no MARKs have been performed, or all have been erased.

This is an atomic command; do not confuse it with the list command (← *PATTERN*).

←← [Editor Command]

Similar to ← but also erases the last MARK, i.e., performs (SETQ MARKLST (CDR MARKLST)).

Note that if the user has two chains marked, and wishes to return to the first chain, he must perform ←←, which removes the second mark, and then ←. However, the second mark is then no longer accessible. If the user wants to be able to return to either of two (or more) chains, he can use the following generalized MARK:

(MARK *LITATOM*) [Editor Command]

Sets *LITATOM* to the current edit chain,

(\ *LITATOM*) [Editor Command]

Makes the current edit chain become the value of *LITATOM*.

If the user did not prepare in advance for returning to a particular edit chain, he may still be able to return to that chain with a single command by using \ or \P.

\ [Editor Command]

Makes the edit chain be the value of UNFIND. Generates an error if UNFIND=NIL.

UNFIND is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ↑, ←, ←←, !NX, all commands that involve a search, e.g., F, LC, .., BELOW, et al and \ and \P themselves. One exception is that UNFIND is not reset when the current edit chain is the top level expression, since this could always be returned to via the ↑ command.

For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR, etc.

\P [Editor Command]

Restores the edit chain to its state as of the last print operation, i.e., P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

For example, if the user types P followed by 3 2 1 P, \P will return to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take him back to the second P, i.e., the user could use \P to flip back and forth between the two edit chains.

Note that if the user had typed P followed by F COND, he could use *either* \ or \P to return to the P, i.e., the action of \ and \P are independent.

S *LITATOM* @ [Editor Command]

Sets *LITATOM* (using SETQ) to the current expression after performing (LC . @). The edit chain is not changed.

Thus (S FOO) will set FOO to the current expression, and (S FOO -1 1) will set FOO to the first element in the last element of the current expression.

17.6 COMMANDS THAT MODIFY STRUCTURE

The basic structure modification commands in the editor are:

(*N*) (*N*≥1) [Editor Command]

Deletes the corresponding element from the current expression.

(*N* E_1 ··· E_M) (*N*≥1) [Editor Command]

Replaces the *N*th element in the current expression with E_1 ··· E_M.

(-*N* E_1 ··· E_M) (*N*≥1) [Editor Command]

Inserts E_1 ··· E_M before the *N*th element in the current expression.

(N E_1 ··· E_M) [Editor Command]

Attaches E_1 ··· E_M at the end of the current expression.

As mentioned earlier: *all structure modification done by the editor is destructive, i.e., the editor uses* RPLACA *and* RPLACD *to physically change the structure it was given.* However, all structure modification is undoable, see UNDO (page 17.50).

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than *N* elements. In addition, the command (1), i.e., delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would require changing a list to an atom (i.e., to NIL) which cannot be done. However, the command DELETE will work even if there is only one element in the current expression, since it will ascend to a point where it *can* do the deletion.

If the value of CHANGESARRAY is a hash array, the editor will mark all structures that are changed by doing (PUTHASH *STRUCTURE* *FN* CHANGESARRAY), where *FN* is the name of the function. The algorithm used for marking is as follows: (1) If the expression is inside of another expression already marked as being changed, do nothing. (2) If the change is an insertion of or replacement with a list, mark the list as changed. (3) If the change is an insertion of or replacement with an atom, or a deletion, mark the parent as changed.

CHANGESARRAY is primarily for use by PRETTYPRINT (page 6.47). When the value of CHANGECHAR is non-NIL, PRETTYPRINT, when printing to a file or display terminal, prints CHANGECHAR in the right margin while printing an expression marked as having been changed. CHANGECHAR is initially |.

17.6.1 Implementation of Structure Modification Commands

Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor,[9] *copies* of the corresponding structure are used, because of the possibility that the exact same command, (i.e., same list structure) might be used again. Thus if a program constructs the command (1 (A B C)) e.g., via (LIST 1 FOO), and gives this command to the editor, the (A B C) used for the replacement will *not* be EQ to FOO.[10]

The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these outside pointers. For example, if the value of FOO is CDR of the current expression, what will the commands (2), (3), (2 X Y Z), (-2 X Y Z), etc. do to FOO?

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if FOO is EQ to the current expression which is (A B C D), and FIE is CDR of FOO, after executing the command (1), FOO will be (B C D) (which is EQUAL but not EQ to FIE). However, under the same initial conditions, after executing (2) FIE will be unchanged, i.e., FIE will still be (B C D) even though the current expression and FOO are now (A C D).[11]

Both replacement and insertion are accomplished by smashing both CAR and CDR of the corresponding tail. Thus, if FOO were EQ to the current expression, (A B C D), after (1 X Y Z), FOO would be (X Y Z B C D). Similarly, if FOO were EQ to the current expression, (A B C D), then after (-1 X Y Z), FOO would be (X Y Z A B C D).

The N command is accomplished by smashing the last CDR of the current expression a la NCONC. Thus if FOO were EQ to any tail of the current expression, after executing an N command, the corresponding expressions would also appear at the end of FOO.

In summary, the only situation in which an edit operation will *not* change an external pointer occurs when the external pointer is to a *proper tail* of the data structure, i.e., to CDR of some node in the structure, and the operation is deletion. If all external pointers are to *elements* of the structure, i.e., to CAR of some

[9] Some editor commands take as arguments a list of edit commands, e.g., (LP F FOO (1 (CAR FOO))). In this case, the command (1 (CAR FOO)) is not considered to have been "typed in" even though the LP command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to EDITF, EDITV, et al, e.g., EDITF(FOO F COND (N --)) are not considered typed in.

[10] The user can circumvent this by using the I command (page 17.45), which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of FOO itself.

[11] A general solution of the problem just isn't possible, as it would require being able to make two lists EQ to each other that were originally different. Thus if FIE is CDR of the current expression, and FUM is CDDR of the current expression, performing (2) would have to make FIE be EQ to FUM if all subsequent operations were to update both FIE and FUM correctly.

node, or if only insertions, replacements, or attachments are performed, the edit operation will *always* have the same effect on an external pointer as it does on the current expression.

17.6.2 The A, B, and : Commands

In the (*N*), (*N* E_1 ··· E_M), and (-*N* E_1 ··· E_M) commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element, (hence the necessity for a separate **N** command). Similarly, the user cannot specify deletion or replacement of the *N*th element from the end of a list without first converting *N* to the corresponding positive integer. Accordingly, we have:

(B E_1 ··· E_M) [Editor Command]

Inserts E_1 ··· E_M before the current expression. Equivalent to UP followed by (-1 E_1 ··· E_M).

For example, to insert FOO before the last element in the current expression, perform -1 and then (B FOO).

(A E_1 ··· E_M) [Editor Command]

Inserts E_1 ··· E_M after the current expression. Equivalent to UP followed by (-2 E_1 ··· E_M) or (N E_1 ··· E_M), whichever is appropriate.

(: E_1 ··· E_M) [Editor Command]

Replaces the current expression by E_1 ··· E_M. Equivalent to UP followed by (1 E_1 ··· E_M).

DELETE [Editor Command]
(:) [Editor Command]

Deletes the current expression.

DELETE first tries to delete the current expression by performing an UP and then a (1). This works in most cases. However, if after performing UP, the new current expression contains only one element, the command (1) will not work. Therefore, DELETE starts over and performs a BK, followed by UP, followed by (2). For example, if the current expression is (COND ((MEMB X Y)) (T Y)), and the user performs -1, and then DELETE, the BK-UP-(2) method is used, and the new current expression will be ... ((MEMB X Y))).

However, if the next higher expression contains only one element, BK will not work. So in this case, DELETE performs UP, followed by (: NIL), i.e., it *replaces* the higher expression by NIL. For example, if the current expression is (COND ((MEMB X Y)) (T Y)) and the user performs F MEMB and then DELETE, the new current expression will be ... NIL (T Y)) and the original expression would now be (COND NIL (T Y)). The rationale behind this is that deleting (MEMB X Y) from ((MEMB X Y)) changes a list of one element to a list of no elements, i.e., () or NIL.

If the current expression is a tail, then B, A, :, and DELETE all work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (B (PRINT X)) would insert (PRINT X) before (PRINT Y), leaving the current expression ... (PRINT X) (PRINT Y) (PRINT Z)).

The following forms of the A, B, and : commands incorporate a location specification:

(INSERT $E_1 \cdots E_M$ BEFORE . @) [Editor Command]

(@ is (CDR (MEMBER 'BEFORE *COMMAND*))) Similar to (LC .@) followed by (B $E_1 \cdots E_M$).

Warning: If @ causes an error, the location process does *not* continue as described on page 17.17. For example if @=(COND 3) and the next COND does not have a 3rd element, the search stops and the INSERT fails. Note that the user can always write (LC COND 3) if he intends the search to continue.

```
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) (PRIN1 & T)
(PRIN1 & T) (SETQ X &

*(INSERT LABEL BEFORE PRIN1)
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) LABEL
(PRIN1 & T) (         user typed control-E

*
```

Current edit chain is not changed, but UNFIND is set to the edit chain after the B was performed, i.e., \ will make the edit chain be that chain where the insertion was performed.

(INSERT $E_1 \cdots E_M$ AFTER . @) [Editor Command]

Similar to INSERT BEFORE except uses A instead of B.

(INSERT $E_1 \cdots E_M$ FOR . @) [Editor Command]

Similar to INSERT BEFORE except uses : for B.

(REPLACE @ BY $E_1 \cdots E_M$) [Editor Command]

(REPLACE @ WITH $E_1 \cdots E_M$) [Editor Command]

Here @ is the *segment* of the command between REPLACE and WITH. Same as (INSERT $E_1 \cdots E_M$ FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO $E_1 \cdots E_M$) [Editor Command]

Same as REPLACE WITH.

(DELETE . @) [Editor Command]

Does a (LC . @) followed by DELETE.[12] The current edit chain is not changed, but UNFIND is set to the edit chain after the DELETE was performed.

Note: the edit chain will be changed if the current expression is no longer a part of the expression being edited, e.g., if the current expression is ... C) and the user performs (DELETE 1), the tail, (C), will have been cut off. Similarly, if the

[12] See warning about INSERT, page 17.25.

current expression is (CDR Y) and the user performs (REPLACE WITH (CAR X)).

Example: (DELETE -1), (DELETE COND 3)

Note: if @ *is* NIL *(i.e., empty), the corresponding operation is performed on the current edit chain.*

For example, (REPLACE WITH (CAR X)) is equivalent to (: (CAR X)). For added readability, HERE is also permitted, e.g., (INSERT (PRINT X) BEFORE HERE) will insert (PRINT X) before the current expression (but not change the edit chain).

Note: @ *does not have to specify a location within the current expression, i.e., it is perfectly legal to ascend to* INSERT, REPLACE, *or* DELETE

For example, (INSERT (RETURN) AFTER ↑ PROG -1) will go to the top, find the first PROG, and insert a (RETURN) at its end, and not change the current edit chain.

The A, B, and : commands, commands, (and consequently INSERT, REPLACE, and CHANGE), all make special checks in E_1 thru E_M for expressions of the form (*##* . *COMS*). In this case, the expression used for inserting or replacing is a *copy* of the current expression after executing *COMS*, a list of edit commands (the execution of *COMS* does not change the current edit chain). For example, (INSERT (*##* F COND -1 -1) AFTER 3) will make a copy of the last form in the last clause of the next COND, and insert it after the third element of the current expression. Note that this is not the same as (INSERT F COND -1 (*##* -1) AFTER 3), which inserts four elements after the third element, namely F, COND, -1, and a copy of the last element in the current expression.

17.6.3 Form Oriented Editing and the Role of UP

The UP that is performed before A, B, and : commands[13] makes these operations form-oriented. For example, if the user types F SETQ, and then DELETE, or simply (DELETE SETQ), he will delete the entire SETQ expression, whereas (DELETE X) if X is a variable, deletes just the variable X. In both cases, the operation is performed on the corresponding *form*, and in both cases is probably what the user intended. Similarly, if the user types (INSERT (RETURN Y) BEFORE SETQ), he means before the SETQ expression, not before the atom SETQ.[14] A consequent of this procedure is that a pattern of the form (SETQ Y --) can be viewed as simply an elaboration and further refinement of the pattern SETQ. Thus (INSERT (RETURN Y) BEFORE SETQ) and (INSERT (RETURN Y) BEFORE (SETQ Y --)) perform the same operation[15] and, in fact, this is one of the motivations behind making the current expression after F SETQ, and F (SETQ Y --) be the same.

Occasionally, however, a user may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as Interlisp attaches to atoms that appear as CAR of a list,

[13] and therefore in INSERT, CHANGE, REPLACE, and DELETE commands after the location portion of the operation has been performed.

[14] There is some ambiguity in (INSERT *EXPR* AFTER *FUNCTIONNAME*), as the user might mean make *EXPR* be the function's first argument. Similarly, the user cannot write (REPLACE SETQ WITH SETQQ) meaning change the name of the function. The user must in these cases write (INSERT *EXPR* AFTER *FUNCTIONNAME* 1), and (REPLACE SETQ 1 WITH SETQQ).

[15] assuming the next SETQ is of the form (SETQ Y --).

versus those appearing elsewhere in a list. In general, the user may not even *know* whether a particular atom is at the head of a list or not. Thus, when he writes (INSERT *EXPR* BEFORE FOO), he means before the atom FOO, whether or not it is CAR of a list. By setting the variable UPFINDFLG to NIL (initially T), the user can suppress the implicit UP that follows searches for atoms, and thus achieve the desired effect. With UPFINDFLG=NIL, following F FOO, for example, the current expression will be the atom FOO. In this case, the A, B, and : operations will operate with respect to the atom FOO. If the user intends the operation to refer to the list which FOO heads, he simply uses instead the pattern (FOO --).

17.6.4 Extract and Embed

Extraction involves replacing the current expression with one of its subexpressions (from any depth).

(XTR . @) [Editor Command]

Replaces the original current expression with the expression that is current after performing (LCL . @).[16] If the current expression after (LCL . @) is a *tail* of a higher expression, its first element is used.

If the extracted expression is a list, then after XTR has finished, the current expression will be that list. If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list.

For example, if the current expression is (COND ((NULL X) (PRINT Y))), (XTR PRINT), or (XTR 2 2) will replace the COND by the PRINT. The current expression after the XTR would be (PRINT Y).

If the current expression is (COND ((NULL X) Y) (T Z)), then (XTR Y) will replace the COND with Y, even though the current expression after performing (LCL Y) is ... Y). The current expression after the XTR would be ... Y followed by whatever followed the COND.

If the current expression *initially* is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus if the current expression is ... (COND ((NULL X) (PRINT Y))) (RETURN Z)), then (XTR PRINT) will replace the COND by the PRINT, leaving (PRINT Y) as the current expression.

The extract command can also incorporate a location specification:

(EXTRACT $@_1$ FROM . $@_2$) [Editor Command]

($@_1$ is the *segment* between EXTRACT and FROM.) Performs (LC . $@_2$)[17] and then (XTR . $@_1$). The current edit chain is not changed, but UNFIND is set to the edit chain after the XTR was performed.

For example: If the current expression is (PRINT (COND ((NULL X) Y) (T Z))) then following (EXTRACT Y FROM COND), the current expression will be (PRINT Y). (EXTRACT 2 -1 FROM COND), (EXTRACT Y FROM 2), and (EXTRACT 2 -1 FROM 2) will all produce the same result.

[16] See warning about INSERT, page 17.25.

[17] See warning about INSERT, page 17.25.

Extract and Embed

While extracting replaces the current expression by a subexpression, embedding replaces the current expression with one containing *it* as a subexpression.

(MBD E_1 ··· E_M) [Editor Command]

MBD substitutes the current expression for all instances of the atom & in E_1 ··· E_M, and replaces the current expression with the result of that substitution. As with SUBST, a fresh copy is used for each substitution.

If & does not appear in E_1 ··· E_M, the MBD is interpreted as (MBD (E_1 ··· E_M &)).

MBD leaves the edit chain so that the larger expression is the new current expression.

Examples:

If the current expression is (PRINT Y), (MBD (COND ((NULL X) &) ((NULL (CAR Y)) & (GO LP)))) would replace (PRINT Y) with (COND ((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP))).

If the current expression is (RETURN X), (MBD (PRINT Y) (AND FLG &)) would replace it with the *two* expressions (PRINT Y) and (AND FLG (RETURN X)) i.e., if the (RETURN X) appeared in the cond clause (T (RETURN X)), after the MBD, the clause would be (T (PRINT Y) (AND FLG (RETURN X))).

If the current expression is (PRINT Y), then (MBD SETQ X) will replace it with (SETQ X (PRINT Y)). If the current expression is (PRINT Y), (MBD RETURN) will replace it with (RETURN (PRINT Y)).

If the current expression *initially* is a tail, embedding works exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (MBD SETQ X) would replace (PRINT Y) with (SETQ X (PRINT Y)).

The embed command can also incorporate a location specification:

(EMBED @ IN . *x*) [Editor Command]

(@ is the segment between EMBED and IN.) Does (LC . @)[18] and then (MBD . *x*). Edit chain is not changed, but UNFIND is set to the edit chain after the MBD was performed.

Examples: (EMBED PRINT IN SETQ X), (EMBED 3 2 IN RETURN), (EMBED COND 3 1 IN (OR & (NULL X))).

WITH can be used for IN, and SURROUND can be used for EMBED, e.g., (SURROUND NUMBERP WITH (AND & (MINUSP X))).

EDITEMBEDTOKEN [Variable]

The special atom used in the MBD and EMBED commands is the value of this variable, initially &.

[18] See warning about INSERT, page 17.25.

17.6.5 The MOVE Command

The MOVE command allows the user to specify (1) the expression to be moved, (2) the place it is to be moved to, and (3) the operation to be performed there, e.g., insert it before, insert it after, replace, etc.

(MOVE @$_1$ TO *COM* . @$_2$) [Editor Command]

(@$_1$ is the segment between MOVE and TO.) *COM* is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. Performs (LC . @$_1$),[19] and obtains the current expression there (or its first element, if it is a tail), which we will call *EXPR*; MOVE then goes back to the original edit chain, performs (LC . @$_2$) followed by (*COM* *EXPR*) (setting an internal flag so *EXPR* is not copied), then goes back to @$_1$ and deletes *EXPR*. The edit chain is not changed. UNFIND is set to the edit chain after (*COM* *EXPR*) was performed.

If @$_2$ specifies a location *inside of the expression to be moved*, a message is printed and an error is generated, e.g., (MOVE 2 TO AFTER X), where X is contained inside of the second element.

For example, if the current expression is (A B C D), (MOVE 2 TO AFTER 4) will make the new current expression be (A C D B). Note that 4 was executed as of the original edit chain, and that the second element had not yet been removed.

As the following examples taken from actual editing will show, the MOVE command is an extremely versatile and powerful feature of the editor.

```
*?
(PROG ((L L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L L)) (RETURN (EDLOC (CDDR C))))
*

*P
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & &))
*(MOVE 2 TO N 1)
*P
... (SELECTQ OBJPR & & &) LP2 (COND & &))

*

*P
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
*(MOVE 4 TO AFTER (BELOW COND))
*P
(OR (EQ X LASTAIL) (NOT &))
*\ P
... (& &) (AND & & &) (T & &))
*
```

[19] See warning about INSERT, page 17.25.

```
*P
((NULL X) **COMMENT** (COND & &))
*(-3 (GO NXT]
*(MOVE 4 TO N (← PROG))
*P
((NULL X) **COMMENT** (GO NXT))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) (COND & &))
*(INSERT NXT BEFORE -1)
*P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) NXT (COND & &))
```

Note that in the last example, the user could have added the PROG label NXT and moved the COND in one operation by performing (MOVE 4 TO N (← PROG) (N NXT)). Similarly, in the next example, in the course of specifying $@_2$, the location where the expression was to be moved to, the user also performs a structure modification, via (N (T)), thus creating the structure that will receive the expression being moved.

```
*P
((CDR &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
*MOVE 4 TO N 0 (N (T)) -1]
*P
((CDR &) **COMMENT** (SETQ CL &))
*\ P
*(T (EDITSMASH CL & &))
*
```

If $@_2$ is NIL, or (HERE), the current position specifies where the operation is to take place. In this case, UNFIND is set to where the expression that was moved was originally located, i.e., $@_1$. For example:

```
*P
(TENEX)
*(MOVE ↑ F APPLY TO N HERE)
*P
(TENEX (APPLY & &))
*

*P
(PROG (& & & ATM IND VAL) (OR & &)   **COMMENT**   (OR & &)
(PRIN1 & T) (
PRIN1 & T) (SETQ IND         user typed control-E

*(MOVE * TO BEFORE HERE)
*P
(PROG (& & & ATM IND VAL) (OR & &) (OR & &) (PRIN1 &

*P
(T (PRIN1 C-EXP T))
*(MOVE ↑ BF PRIN1 TO N HERE)
*P
(T (PRIN1 C-EXP T) (PRIN1 & T))
```

```
*
```

Finally, if $@_1$ is NIL, the MOVE command allows the user to specify where the *current expression* is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; UNFIND is set to where it was.

```
*P
(SELECTQ OBJPR (&) (PROGN & &))
*(MOVE TO BEFORE LOOP)
*P
... (SELECTQ OBJPR & &) LOOP (FRPLACA DFPRP &) (FRPLACD DFPRP
&) (SELECTQ          user typed control-E

*
```

17.6.6 Commands That Move Parentheses

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by PRINT. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens.

In all six commands, *N* and *M* are used to specify an element of a list, usually of the current expression. In practice, *N* and *M* are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized NTH command (NTH *COM*) to find their element(s), so that *N*th element means the first element of the tail found by performing (NTH *N*). In other words, if the current expression is (LIST (CAR X) (SETQ Y (CONS W Z))), then (BI 2 CONS), (BI X -1), and (BI X Z) all specify the exact same operation.

All six commands generate an error if the element is not found, i.e., the NTH fails. All are undoable.

(BI *N* *M*) [Editor Command]

“Both In”. Inserts a left parentheses before the *N*th element and after the *M*th element in the current expression. Generates an error if the *M*th element is not contained in the *N*th tail, i.e., the *M*th element must be “to the right” of the *N*th element.

Example: If the current expression is (A B (C D E) F G), then (BI 2 4) will modify it to be (A (B (C D E) F) G).

(BI *N*) [Editor Command]

Same as (BI *N* *N*).

Example: If the current expression is (A B (C D E) F G), then (BI -2) will modify it to be (A B (C D E) (F) G).

(BO *N*) [Editor Command]

“Both Out”. Removes both parentheses from the *N*th element. Generates an error if *N*th element is not a list.

Example: If the current expression is (A B (C D E) F G), then (BO D) will modify it to be (A B C D E F G).

(LI *N*) [Editor Command]

"Left In". Inserts a left parenthesis before the *N*th element (and a matching right parenthesis at the end of the current expression), i.e. equivalent to (BI *N* -1).

Example: if the current expression is (A B (C D E) F G), then (LI 2) will modify it to be (A (B (C D E) F G)).

(LO *N*) [Editor Command]

"Left Out". Removes a left parenthesis from the *N*th element. *All elements following the Nth element are deleted.* Generates an error if *N*th element is not a list.

Example: If the current expression is (A B (C D E) F G), then (LO 3) will modify it to be (A B C D E).

(RI *N M*) [Editor Command]

"Right In". Inserts a right parenthesis after the *M*th element of the *N*th element. The rest of the *N*th element is brought up to the level of the current expression.

Example: If the current expression is (A (B C D E) F G), (RI 2 2) will modify it to be (A (B C) D E F G). Another way of thinking about RI is to read it as "move the right parenthesis at the end of the *N*th element *in* to after its *N*th element."

(RO *N*) [Editor Command]

"Right Out". Removes the right parenthesis from the *N*th element, moving it to the end of the current expression. All elements following the *N*th element are moved inside of the *N*th element. Generates an error if *N*th element is not a list.

Example: If the current expression is (A B (C D E) F G), (RO 3) will modify it to be (A B (C D E F G)). Another way of thinking about RO is to read it as "move the right parenthesis at the end of the *N*th element *out* to the end of the current expression."

17.6.7 TO and THRU

EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using in their respective location specifications the TO or THRU command.

(@$_1$ THRU @$_2$) [Editor Command]

Does a (LC . @$_1$), followed by an UP, and then a (BI 1 @$_2$), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element.

For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)).

(@$_1$ TO @$_2$) [Editor Command]

Same as THRU except the last element not included, i.e., after the BI, an (RI 1 -2) is performed.

If both $@_1$ and $@_2$ are numbers, and $@_2$ is greater than $@_1$, then $@_2$ counts from the beginning of the current expression, the same as $@_1$. In other words, if the current expression is (A B C D E F G), (3 THRU 5) means (C THRU E) not (C THRU G). In this case, the corresponding BI command is (BI 1 $@_2$-$@_1$+1).

THRU and TO are not very useful commands by themselves; they are intended to be used in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

```
*P
(PROG (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ IND &)
(SETQ VAL &) **COMMENT** (SETQQ       user typed control-E

*(MOVE (3 THRU 4) TO BEFORE 7)
*P
(PROG (& & ATM IND VAL WORD) (SETQ IND &) (SETQ VAL &) (PRIN1 & T)
(PRIN1 & T) **COMMENT**       user typed control-E

*


*P
(* FAIL RETURN FROM EDITOR. USER SHOULD NOTE THE VALUES OF SOURCEXPR
AND CURRENTFORM.  CURRENTFORM IS THE LAST FORM IN SOURCEXPR WHICH WILL
HAVE BEEN TRANSLATED, AND IT CAUSED THE ERROR.)
*(DELETE (USER THRU CURR$))
=CURRENTFORM.
*P
(* FAIL RETURN FROM EDITOR.  CURRENTFORM IS       user typed control-E

*


*P
... LP (SELECTO & & & & NIL) (SETQ Y &) OUT (SETQ FLG &) (RETURN Y))
*(MOVE (1 TO OUT) TO N HERE]
*P
... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & & & NIL) (SETQ Y &))
*


*PP
[PROG (RF TEMP1 TEMP2)
   (COND
      ((NOT (MEMB REMARG LISTING))
         (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS)) **COMMENT**
         (SETQ TEMP2 (CADR TEMP1))
         (GO SKIP))
      (T    **COMMENT**
         (SETQ TEMP1 REMARG)))
   (NCONC1 LISTING REMARG)
   (COND
      ((NOT (SETQ TEMP2 (SASSOC
*(EXTRACT (SETQ THRU CADR) FROM COND)
*P
(PROG (RF TEMP1 TEMP2) (SETQ TEMP1 &) **COMMENT** (SETQ TEMP2 &) (NCONC1 LISTING
REMARG) (COND & &      user typed control-E

*
```

TO and THRU can also be used directly with XTR, because XTR involves a location specification while A, B, :, and MBD do not. Thus in the previous example, if the current expression had been the COND, e.g., the user had first performed F COND, he could have used (XTR (SETQ THRU CADR)) to perform the extraction.

(@$_1$ TO) [Editor Command]

(@$_1$ THRU) [Editor Command]

Both are the same as (@$_1$ THRU -1), i.e., from @$_1$ through the end of the list.

Examples:

```
*P
(VALUE (RPLACA DEPRP &) (RPLACD &) (RPLACA VARSWORD &) (RETURN))
*(MOVE (2 TO) TO N (← PROG))
*(N (GO VAR))
*P
(VALUE (GO VAR))

*P
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*(-3 (GO REPLACE))
*(MOVE (COND TO) TO N ↑ PROG (N REPLACE))
*P
(T **COMMENT** (GO REPLACE))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) DELETE (COND & &) REPLACE
(COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*

*PP
[LAMBDA (CLAUSALA X)
   (PROG (A D)
      (SETQ A CLAUSALA)
  LP  (COND
         ((NULL A)
            (RETURN)))
      (SERCH X A)
      (RUMARK (CDR A))
      (NOTICECL (CAR A))
      (SETQ A (CDR A))
      (GO LP]
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
=NOTICECL
*P
```

```
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL &))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA (A) *]
*PP
[LAMBDA (CLAUSALA X)
   (MAP CLAUSALA
        (FUNCTION (LAMBDA (A)
                     (SERCH X A)
                     (RUMARK (CDR A))
                     (NOTICECL (CAR A]
*
```

17.6.8 The R Command

(R *x* *y*) [Editor Command]

Replaces all instances of *x* by *y* in the current expression, e.g., (R CAADR CADAR). Generates an error if there is not at least one instance.

The R command operates in conjunction with the search mechanism of the editor. The search proceeds as described on page 17.15, and *x* can employ any of the patterns on page 17.13. Each time *x* matches an element of the structure, the element is replaced by (a copy of) *y*; each time *x* matches a tail of the structure, the tail is replaced by (a copy of) *y*.

For example, if the current expression is (A (B C) (B . C)),

(R C D) will change it to (A (B D) (B . D)),

(R (... . C) D) will change it to (A (B C) (B . D)),

(R C (D E)) will change it to (A (B (D E)) (B D E)), and

(R (... . NIL) D) will change it to (A (B C . D) (B . C) . D).

If *x* is an atom or string containing $s (<esc>s), $s appearing in *y* stand for the characters matched by the corresponding $ in *x*. For example, (R FOO$ FIE$) means for all atoms or strings that begin with FOO, replace the characters "FOO" by "FIE".[20] Applied to the list (FOO FOO2 XFOO1), *(R FOO$ FIE$)* would produce (FIE FIE2 XFOO1), and (R FOO FIE) would produce (FIE FIE2 XFIE1). Similarly, (R D A) will change (LIST (CADR X) (CADDR Y)) to (LIST (CAAR X) (CAADR)). Note that CADDR was *not* changed to CAAAR, i.e., (R D A) does not mean replace every D with A, but replace the first D in every atom or string by A. If the user wanted to replace every D by A, he could perform (LP (R D A)).

The user will be informed of all such $ replacements by a message of the form *x*->*y*, e.g., CADR->CAAR.

Note that the $ feature can be used to delete or add characters, as well as replace them. For example, (R $1 $) will delete the terminating 1's from all literal atoms and strings. Similarly, if an $ in *x* does

[20] If *x* matches a string, it will be replaced by a string. Note that it does not matter whether *x* or *y* themselves are strings, i.e. (R D A), (R "D" A), (R D "A"), and (R "D" "A") are equivalent. Note also that *x* will never match with a number, i.e., (R $1 $2) will not change 11 to 12.

not have a mate in *Y*, the characters matched by the $ are effectively deleted. For example, (R $/$ $) will change AND/OR to AND.[21] *Y* can also be a list containing $s, e.g., (R $1 (CAR $)) will change FOO1 to (CAR FOO), FIE1 to (CAR FIE).

If *X* does not contain $s, $ appearing in *Y* refers to the *entire* expression matched by *X*, e.g., (R LONGATOM '$) changes LONGATOM to 'LONGATOM, (R (SETQ X &) (PRINT $)) changes every (SETQ X &) to (PRINT (SETQ X &)).[22]

Since (R $*X*$ $*Y*$) is a frequently used operation for Replacing Characters, the following command is provided:

(RC *X* *Y*) [Editor Command]

Equivalent to (R $*X*$ $*Y*$)

R and RC change all instances of *X* to *Y*. The commands R1 and RC1 are available for changing just one, (i.e., the first) instance of *X* to *Y*.

(R1 *X* *Y*) [Editor Command]

Find the first instance of *X* and replace it by *Y*.

(RC1 *X* *Y*) [Editor Command]

(R1 $*X*$ $*Y*$).

In addition, while R and RC only operate within the current expression, R1 and RC1 will continue searching, a la the F command, until they find an instance of *x*, even if the search carries them beyond the current expression.

(SW *N* *M*) [Editor Command]

Switches the *N*th and *M*th elements of the current expression.

For example, if the current expression is (LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (CDR Y))), (SW 2 3) will modify it to be (LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y))). The relative order of *N* and *M* is not important, i.e., (SW 3 2) and (SW 2 3) are equivalent.

SW uses the generalized NTH command (NTH *COM*) to find the *N*th and *M*th elements, a la the BI-BO commands.

Thus in the previous example, (SW CAR CDR) would produce the same result.

(SWAP $@_1$ $@_2$) [Editor Command]

Like SW except switches the expressions specified by $@_1$ and $@_2$, not the corresponding elements of the current expression, i.e. $@_1$ and $@_2$ can be at different levels in current expression, or one or both be outside of current expression.

[21] There is no similar operation for changing AND/OR to OR, since the first $ in *Y* always corresponds to the first $ in *X*, the second $ in *Y* to the second in *X*, etc.

[22] If *X* is a pattern containing an $ pattern somewhere *within* it, the characters matched by the $s are not available, and for the purposes of replacement, the effect is the same as though *X* did not contain any $s. For example, if the user types (R (CAR F$) (PRINT $)), the second $ will refer to the entire expression matched by (CAR F$).

Thus, using the previous example, (SWAP CAR CDR) would result in (LIST (CONS (CDR X) (CAR Y)) (CONS (CAR X) (CDR Y))).

17.7 COMMANDS THAT PRINT

PP [Editor Command]

Prettyprints the current expression.

P [Editor Command]

Prints the current expression as though PRINTLEVEL (page 6.18) were set to 2.

(P *M*) [Editor Command]

Prints the *M*th element of the current expression as though PRINTLEVEL were set to 2.

(P 0) [Editor Command]

Same as P.

(P *M N*) [Editor Command]

Prints the *M*th element of the current expression as though PRINTLEVEL were set to *N*.

(P 0 *N*) [Editor Command]

Prints the current expression as though PRINTLEVEL were set to *N*.

? [Editor Command]

Same as (P 0 100).

Both (P *M*) and (P *M N*) use the generalized NTH command (NTH *COM*) to obtain the corresponding element, so that *M* does not have to be a number, e.g., (P COND 3) will work. PP causes all comments to be printed as **COMMENT** (see page 6.50). P and ? print as **COMMENT** only those comments that are (top level) elements of the current expression. Lower expressions are not really seen by the editor; the printing command simply sets PRINTLEVEL and calls PRINT.

PP* [Editor Command]

Prettyprints current expression, *including* comments.

PP* is equivalent to PP except that it first resets **COMMENT**FLG to NIL (see page 6.50).

PPV [Editor Command]

Prettyprints the current expression as a variable, i.e., no special treatment for LAMBDA, COND, SETQ, etc., or for CLISP.

PPT [Editor Command]

Prettyprints the current expression, printing CLISP translations, if any.

?= [Editor Command]

Prints the argument names and corresponding values for the current expression. Analagous to the ?= break command (page 9.5). For example,

```
*P
(STRPOS "A0???" X N (QUOTE ?) T)
*?=
X = "A0???"
Y = X
START = N
SKIP = (QUOTE ?)
ANCHOR = T
TAIL =
```

The command MAKE (page 17.44) is an imperative form of ?=. It allows the user to specify a change to the element of the current expression that corresponds to a particular argument name.

All printing functions print to the terminal, regardless of the primary output file. All use the readtable T. No printing function ever changes the edit chain. All record the current edit chain for use by \P (page 17.21). All can be aborted with control-E.

17.8 COMMANDS FOR LEAVING THE EDITOR

OK [Editor Command]

Exits from the editor.

STOP [Editor Command]

Exits from the editor with an error. Mainly for use in conjunction with TTY: commands (page 17.40) that the user wants to abort.

Since all of the commands in the editor are errorset protected, the user must exit from the editor via a command. STOP provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session. For example, if the user is executing (MOVE 3 TO AFTER COND TTY:), and he exits from the lower editor with an OK, the MOVE command will then complete its operation. If the user wants to abort the MOVE command, he must make the TTY: command generate an error. He does this by exiting from the lower editor with a STOP command. In this case, the higher editor's edit chain will not be changed by the TTY: command.

Actually, it is also possible to exit the editor by typing control-D. STOP is preferred even if the user is editing at the EVALQT level, as it will perform the necessary "wrapup" to insure that the changes made while editing will be undoable.

SAVE [Editor Command]

Exits from the editor and saves the "state of the edit" on the property list of the function or variable being edited under the property EDIT-SAVE. If the editor is called again on the same structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of UNFIND and UNDOLST are restored.

For example:

```
*P
(NULL X)
*F COND P
```

```
(COND (& &) (T &))
*SAVE
FOO
← .
  .
  .
←EDITF(FOO)
EDIT
*P
(COND (& &) (T &))
*\ P
(NULL X)
*
```

SAVE is necessary only if the user is editing many different expressions; an exit from the editor via OK always saves the state of the edit of that call to the editor on the property list of the atom EDIT, under the property name LASTVALUE. OK also remprops EDIT-SAVE from the property list of the function or variable being edited.

Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark list and UNDOLST, and sets UNFIND to be the edit chain as of the previous exit from the editor. For example:

```
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
  .
  .
  .
*P
(COND & &)
*OK
FOO
← .
  .                         any number of LISPX inputs
  .                         except for calls to the editor
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
*\ P
(COND & &)
*
```

Furthermore, as a result of the history feature, if the editor is called on the same expression within a certain number of LISPX inputs,[23] the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime. For example:

[23] Namely, the size of the history list, which can be changed with CHANGESLICE, (page 8.18).

```
←EDITF(FOO)
EDIT
*
   .
   .
   .
*P
(COND (& &) (& &) (&) (T &))
*OK
FOO
   .                    a small number of LISPX inputs,
   .                    including editing
   .
←EDITF(FOO)
EDIT
*\ P
(COND (& &) (& &) (&) (T &))
*
```

Thus the user can always continue editing, including undoing changes from a previous editing session, if (1) No other expressions have been edited since that session (since saving takes place at *exit* time, intervening calls that were aborted via control-D or exited via STOP will not affect the editor's memory); or (2) That session was "sufficiently" recent; or (3) It was ended with a SAVE command.

17.9 NESTED CALLS TO EDITOR

TTY: [Editor Command]

Calls the editor recursively. The user can then type in commands, and have them executed. The TTY: command is completed when the user exits from the lower editor. (see OK and STOP above).

The TTY: command is extremely useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. For example the command (MOVE 3 TO AFTER COND 3 P TTY:) allows the user to interact, in effect, *within* the MOVE command. Thus he can verify for himself that the correct location has been found, or complete the specification "by hand." In effect, TTY: says "I'll tell you what you should do when you get there."

The TTY: command operates by printing TTY: and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the TTY: command was entered. Until the user exits from the lower editor, any attention changing commands he executes only affect the lower editor's edit chain. Of course, if the user performs any structure modification commands while under a TTY: command, these will modify the structure in both editors, since it is the same structure. When the TTY: command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

EF [Editor Command]
EV [Editor Command]
EP [Editor Command]

Calls EDITF or EDITV or EDITP on CAR of current expression.

17.10 MANIPULATING THE CHARACTERS OF AN ATOM OR STRING

RAISE [Editor Command]

An edit macro defined as UP followed by (I 1 (U-CASE (## 1))), i.e., it raises to upper-case the current expression, or if a tail, the first element of the current expression.

LOWER [Editor Command]

Similar to RAISE, except uses L-CASE.

CAP [Editor Command]

First does a RAISE, and then lowers all but the first character, i.e., the first character is left capitalized.

Note: RAISE, LOWER, and CAP are all no-ops if the corresponding atom or string is already in that state.

(RAISE *x*) [Editor Command]

Equivalent to (I R (L-CASE *x*) *x*), i.e., changes every lower-case *x* to upper-case in the current expression.

(LOWER *x*) [Editor Command]

Similar to RAISE, except performs (I R *x* (L-CASE *x*)).

Note that in both (RAISE *x*) and (LOWER *x*), *x* should be typed in upper case.

REPACK [Editor Command]

Permits the "editing" of an atom or string.

REPACK operates by calling the editor recursively on UNPACK of the current expression, or if it is a list, on UNPACK of its first element. If the lower editor is exited successfully, i.e., via OK as opposed to STOP, the list of atoms is made into a single atom or string, which replaces the atom or string being "repacked." The new atom or string is always printed.

Example:

```
*P
... "THIS IS A LOGN STRING")
*REPACK
*EDIT
P
(T H I S %  I S %  A %  L O G N %  S T R I N G)
*(SW G N)
*OK
"THIS IS A LONG STRING"
*
```

Note that this could also have been accomplished by (R GN NG) or simply (RC GN NG).

(REPACK @) [Editor Command]

Does (LC . @) followed by REPACK, e.g. (REPACK THIS$).

17.11 MANIPULATING PREDICATES AND CONDITIONAL EXPRESSIONS

JOINC [Editor Command]

Used to join two neighboring COND's together, e.g. (COND *CLAUSE*$_1$ *CLAUSE*$_2$) followed by (COND *CLAUSE*$_3$ *CLAUSE*$_4$) becomes (COND *CLAUSE*$_1$ *CLAUSE*$_2$ *CLAUSE*$_3$ *CLAUSE*$_4$). JOINC does an (F COND T) first so that you don't have to be at the first COND.

(SPLITC *x*) [Editor Command]

Splits one COND into two. *x* specifies the last clause in the first COND, e.g. (SPLITC 3) splits (COND *CLAUSE*$_1$ *CLAUSE*$_2$ *CLAUSE*$_3$ *CLAUSE*$_4$) into (COND *CLAUSE*$_1$ *CLAUSE*$_2$) (COND *CLAUSE*$_3$ *CLAUSE*$_4$). Uses the generalized NTH command (NTH *COM*), so that *x* does not have to be a number, e.g., the user can say (SPLITC RETURN), meaning split after the clause containing RETURN. SPLITC also does an (F COND T) first.

NEGATE [Editor Command]

Negates the current expression, i.e. performs (MBD NOT), except that is smart about simplifying. For example, if the current expression is: (OR (NULL X) (LISTP X)), NEGATE would change it to (AND X (NLISTP X)).

NEGATE is implemented via the function NEGATE (page 14.2).

SWAPC [Editor Command]

Takes a conditional expression of the form (COND (A B)(T C)) and rearranges it to an equivalent (COND ((NOT A) C)(T B)), or (COND (A B) (C D)) to (COND ((NOT A) (COND (C D))) (T B)).

SWAPC is smart about negations (uses NEGATE) and simplifying CONDs. It always produces an equivalent expression. It is useful for those cases where one wants to insert extra clauses or tests.

17.12 HISTORY COMMANDS IN THE EDITOR

As described on page 8.35, all of the user's inputs to the editor are stored on EDITHISTORY, the editor's history list, and all of the programmer's assistant commands for manipulating the history list, e.g. REDO, USE, FIX, NAME, etc., are available for use on events on EDITHISTORY. In addition, the following four history commands are recognized specially by the editor. They always operate on the last, i.e. most recent, event.

DO *COM* [Editor Command]

Allows the user to supply the command name when it was omitted.

USE is useful when a command name is *incorrect*.

For example, suppose the user wants to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon the user can type DO -2. The effect is the same as though the user had typed FIX, followed by (LI 1), (-1 -2), and OK, i.e., the command (-2 (SETQ X (LIST Y Z))) is executed. DO also works if the command is a line command.

!F [Editor Command]

Same as DO F.

In the case of !F, the previous command is always treated as though it were a line command, e.g., if the user types (SETQ X'&) and then !F, the effect is the same as though he had typed F (SETQ X &), not (F (SETQ X &)).

!E [Editor Command]

Same as DO E.

!N [Editor Command]

Same as DO N.

17.13 MISCELLANEOUS COMMANDS

NIL [Editor Command]

Unless preceded by F or BF, is always a no-op. Thus extra right parentheses or square brackets at the ends of commands are ignored.

CL [Editor Command]

Clispifies the current expression (see page 16.17).

DW [Editor Command]

Dwimifies the current expression (see page 16.14).

GET* [Editor Command]

If the current expression is a comment pointer (see page 6.51), reads in the full text of the comment, and replaces the current expression by it.

(* . *x*) [Editor Command]

x is the text of a comment. * ascends the edit chain looking for a "safe" place to insert the comment, e.g., in a COND clause, after a PROG statement, etc., and inserts (* . *x*) *after* that point, if possible, otherwise before. For example, if the current expression is (FACT (SUB1 N)) in

```
[COND
   ((ZEROP N) 1)
   (T (ITIMES N (FACT (SUB1 N]
```

(* CALL FACT RECURSIVELY) would insert (* CALL FACT RECURSIVELY) *before* the ITIMES expression.[24]

[24] If inserted after the ITIMES, the comment would then be (incorrectly) returned as the value of the COND. However, if the COND was itself a PROG statement, and hence its value was not being used, the comment could be (and would be) inserted after the ITIMES expression.

* does not change the edit chain, but UNFIND is set to where the comment was actually inserted.

GETD [Editor Command]

Essentially "expands" the current expression in line: (1) if (CAR of) the current expression is the name of a macro, expands the macro in line; (2) if a CLISP word, translates the current expression and replaces it with the translation; (3) if CAR is the name of a function for which the editor can obtain a symbolic definition, either in-core or from a file, substitutes the argument expressions for the corresponding argument names in the body of the definition and replaces the current expression with the result; (4) if CAR of the current expression is an open lambda, substitutes the arguments for the corresponding argument names in the body of the lambda, and then removes the lambda and argument list.

(MAKEFN (*FN* . *ACTUALARGS*) *ARGLIST* N_1 N_2) [Editor Command]

The inverse of GETD: makes the current expression into a function. *FN* is the function name, *ARGLIST* its arguments. The argument names are substituted for the corresponding argument values in *ACTUALARGS*, and the result becomes the body of the function definition for *FN*. The current expression is then replaced with (*FN* . *ACTUALARGS*).

If N_1 and N_2 are supplied, (N_1 THRU N_2) is used rather than the current expression; if just N_1 is supplied, (N_1 THRU -1) is used.

If *ARGLIST* is omitted, MAKEFN will make up some arguments, using elements of *ACTUALARGS*, if they are literal atoms, otherwise arguments selected from (X Y Z A B C ...), avoiding duplicate argument names.

Example: If the current expression is (COND ((CAR X) (PRINT Y T)) (T (HELP))), then (MAKEFN (FOO (CAR X) Y) (A B)) will define FOO as (LAMBDA (A B) (COND (A (PRINT B T)) (T (HELP)))) and then replace the current expression with (FOO (CAR X) Y).

(MAKE *ARGNAME* *EXP*) [Editor Command]

Makes the value of *ARGNAME* be *EXP* in the call which is the current expression, i.e. a ?= command following a MAKE will always print *ARGNAME*=*EXP*. For example:

```
*P
(JSYS)
*?=
JSYS[N;AC1,AC2,AC3,RESULTAC]
*(MAKE N 10)
*(MAKE RESULTAC 3)
*P
(JSYS 10 NIL NIL NIL 3)
```

Q [Editor Command]

Quotes the current expression, i.e. MBD QUOTE.

D [Editor Command]

Deletes the current expression, then prints new current expression, i.e. (:) I P.

17.14 COMMANDS THAT EVALUATE

E [Editor Command]

Causes the editor to call the Interlisp executive LISPX giving it the next input as argument. Example:

```
*E BREAK(FIE FUM)
(FIE FUM)
*E (FOO)

(FIE BROKEN)
:
```

Note: E only works when when typed in, e.g, (INSERT D BEFORE E) will treat E as a pattern, and search for E.

(E *x*) [Editor Command]

Evaluates *x*, i.e., performs (EVAL *x*), and prints the result on the terminal.

(E *x* T) [Editor Command]

Same as (E *x*) but does not print.

The (E *x*) and (E *x* T) commands are mainly intended for use by macros and subroutine calls to the editor; the user would probably type in a form for evaluation using the more convenient format of the (atomic) E command.

(I *c* x_1 ··· x_N) [Editor Command]

Executes the *editor command* (*c* y_1 ··· y_N) where y_i = (EVAL x_i). If *c* is not an atom, *c* is evaluated also.

Examples:

(I 3 (GETD 'FOO)) will replace the 3rd element of the current expression with the definition of FOO.

(I N FOO (CAR FIE)) will attach the value of FOO and CAR of the value of FIE to the end of the current expression.

(I F= FOO T) will search for an expression EQ to the value of FOO.

(I (COND ((NULL FLG) '-1) (T 1)) FOO), if FLG is NIL, inserts the value of FOO before the first element of the current expression, otherwise replaces the first element by the value of FOO.

The I command sets an internal flag to indicate to the structure modification commands *not* to copy expression(s) when inserting, replacing, or attaching.

EVAL [Editor Command]

Does an EVAL of the current expression.

Note that EVAL, line-feed, and the GO command together effectively allow the user to "single-step" a program through its symbolic definition.

GETVAL [Editor Command]

Replaces the current expression by the result of evaluating it.

(## COM_1 COM_2 ⋯ COM_N) [NLambda NoSpread Function]

An nlambda, nospread function (not a command). Its value is what the current expression would be after executing the edit commands COM_1 ⋯ COM_N starting from the present edit chain. Generates an error if any of COM_1 thru COM_N cause errors. The current edit chain is never changed.[25]

Example: (I R 'X (## (CONS .. Z))) replaces all X's in the current expression by the first CONS containing a Z.

The I command is not very convenient for computing an *entire* edit command for execution, since it computes the command name and its arguments separately. Also, the I command cannot be used to compute an atomic command. The following two commands provide more general ways of computing commands.

(COMS x_1 ⋯ x_M) [Editor Command]

Each x_i is evaluated and its value is executed as a command.

For example, (COMS (COND (X (LIST 1 X)))) will replace the first element of the current expression with the value of X if non-NIL, otherwise do nothing.[26]

(COMSQ COM_1 ⋯ COM_N) [Editor Command]

Executes COM_1 ⋯ COM_N.

COMSQ is mainly useful in conjunction with the COMS command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the COMS command. He would then write (COMS (CONS 'COMSQ *x*)) where *x* computed the list of commands, e.g., (COMS (CONS 'COMSQ (GETP FOO 'COMMANDS))).

17.15 COMMANDS THAT TEST

(IF *x*) [Editor Command]

Generates an error *unless* the value of (EVAL *x*) is true. In other words, if (EVAL *x*) causes an error or (EVAL *x*)=NIL, IF will cause an error.

For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on, as COND uses NIL and non-NIL. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location specification (IPLUS (E (OR (NUMBERP (## 3)) (ERROR!)) T)) specifies the first IPLUS whose second argument is a number. The IF command, by equating NIL to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is (IPLUS (IF (NUMBERP (## 3)))).

[25] The A, B, :, INSERT, REPLACE, and CHANGE commands make special checks for ## forms in the expressions used for inserting or replacing, and use a copy of ## form instead (see page 17.26). Thus, (INSERT (## 3 2) AFTER 1) is equivalent to (I INSERT (COPY (## 3 2)) 'AFTER 1).

[26] The editor command NIL is a no-op, see page 17.43.

The IF command can also be used to select between two alternate lists of commands for execution.

(IF *x* *coms*$_1$ *coms*$_2$) [Editor Command]

If (EVAL *x*) is true, execute *coms*$_1$; if (EVAL *x*) causes an error or is equal to NIL, execute *coms*$_2$.

Thus IF is equivalent to

```
(COMS (CONS 'COMSQ
            (COND
               ((CAR (NLSETQ (EVAL X)))
                  coms1)
               (T  coms2))))
```

For example, the command (IF (READP T) NIL (P)) will print the current expression provided the input buffer is empty.

(IF *x* *coms*$_1$) [Editor Command]

If (EVAL *x*) is true, execute *coms*$_1$; otherwise generate an error.

(LP *coms*$_1$ ··· *coms*$_N$) [Editor Command]

Repeatedly executes *coms*$_1$ ··· *coms*$_N$ until an error occurs.

For example, (LP F PRINT (N T)) will attach a T at the end of every PRINT expression. (LP F PRINT (IF (## 3) NIL ((N T)))) will attach a T at the end of each print expression which does not already have a second argument.[27]

When an error occurs, LP prints *N* OCCURRENCES where *N* is the number of times the commands were successfully executed. The edit chain is left as of the last complete successful execution of *coms*$_1$ ··· *coms*$_N$.

(LPQ *coms*$_1$ ··· *coms*$_N$) [Editor Command]

Same as LP but does not print the message *N* OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches MAXLOOP, initially set to 30. MAXLOOP can be set to NIL, which is equivalent to setting it to infinity. Since the edit chain is left as of the last successful completion of the loop, the user can simply continue the LP command with REDO (page 8.7).

(SHOW *x*) [Editor Command]

x is a list of patterns. SHOW does a LPQ printing all instances of the indicated expression(s), e.g. (SHOW FOO (SETQ FIE &)) will print all FOO's and all (SETQ FIE &)'s. Generates an error if there aren't any instances of the expression(s).

[27] The form (## 3) will cause an error if the edit command 3 causes an error, thereby selecting ((N T)) as the list of commands to be executed. The IF could also be written as (IF (CDDR (##)) NIL ((N T))).

`(EXAM x)` [Editor Command]

Like `SHOW` except calls the editor recursively (via the `TTY:` command, see page 17.40) on each instance of the indicated espression(s) so that the user can examine and/or change them.

`(ORR` *COMS*$_1$ ··· *COMS*$_N$`)` [Editor Command]

`ORR` begins by executing *COMS*$_1$, a list of commands. If no error occurs, `ORR` is finished. Otherwise, `ORR` restores the edit chain to its original value, and continues by executing *COMS*$_2$, etc. If none of the command lists execute without errors, i.e., the `ORR` "drops off the end", `ORR` generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without an error.

`NIL` as a command list is perfectly legal, and will always execute successfully. Thus, making the last "argument" to `ORR` be `NIL` will insure that the `ORR` never causes an error. Any other atom is treated as `(`*ATOM*`)`, i.e., the above example could be written as `(ORR NX !NX NIL)`.

For example, `(ORR (NX) (!NX) NIL)` will perform a `NX`, if possible, otherwise a `!NX`, if possible, otherwise do nothing. Similarly, `DELETE` could be written as `(ORR (UP (1)) (BK UP (2)) (UP (: NIL)))`.

17.16 EDIT MACROS

Many of the more sophisticated branching commands in the editor, such as `ORR`, `IF`, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire, or redefine existing commands.[28] Macros are defined by using the `M` command:

`(M` *C* *COMS*$_1$ ··· *COMS*$_N$`)` [Editor Command]

For *C* an atom, `M` defines *C* as an atomic command. If a macro is redefined, its new definition replaces its old. Executing *C* is then the same as executing the list of commands *COMS*$_1$ ··· *COMS*$_N$.

For example, `(M BP BK UP P)` will define `BP` as an atomic command which does three things, a `BK`, and `UP`, and a `P`. Macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose `Z` is defined by `(M Z -1 (IF (READP T) NIL (P)))`, i.e., `Z` does a `-1`, and then if nothing has been typed, a `P`. Now we can define `ZZ` by `(M ZZ -1 Z)`, and `ZZZ` by `(M ZZZ -1 -1 Z)` or `(M ZZZ -1 ZZ)`.

Macros can also define list commands, i.e., commands that take arguments.

`(M (`*C*`) (`*ARG*$_1$ ··· *ARG*$_N$`)` *COMS*$_1$ ··· *COMS*$_M$`)` [Editor Command]

C an atom. `M` defines *C* as a list command. Executing `(`*C* *E*$_1$ ··· *E*$_N$`)` is then performed by substituting *E*$_1$ for *ARG*$_1$, ··· *E*$_N$ for *ARG*$_N$ throughout *COMS*$_1$ ··· *COMS*$_M$, and then executing *COMS*$_1$ ··· *COMS*$_M$.

[28] To refer to the original definition of a built-in command when redefining it via a macro, use the `ORIGINAL` command (page 17.50).

For example, we could define a more general BP by (M (BP) (N) (BK N) UP P). Thus, (BP 3) would perform (BK 3), followed by an UP, followed by a P.

A list command can be defined via a macro so as to take a fixed or indefinite number of "arguments", as with spread vs. nospread functions. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the "argument list" is *atomic*, the command takes an indefinite number of arguments.

(M (*C*) *ARG* $COMS_1 \cdots COMS_M$) [Editor Command]

If *C*, *ARG* are both atoms, this defines *C* as a list command. Executing (*C* $E_1 \cdots E_N$) is performed by substituting ($E_1 \cdots E_N$), i.e., CDR of the command, for *ARG* throughout $COMS_1 \cdots COMS_M$, and then executing $COMS_1 \cdots COMS_M$.

For example, the command 2ND (page 17.18), could be defined as a macro by (M (2ND) X (ORR ((LC . X) (LC . X)))).

Note that for all editor commands, "built in" commands as well as commands defined by macros as atomic commands and list definitions are *completely* independent. In other words, the existence of an atomic definition for *C* in *no* way affects the treatment of *C* when it appears as CAR of a list command, and the existence of a list definition for *C* in *no* way affects the treatment of *C* when it appears as an atom. In particular, *C* can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used.

Note also that once *C* is defined as an atomic command via a macro definition, it will *not* be searched for when used in a location specification, unless it is preceded by an F. Thus (INSERT -- BEFORE BP) would not search for BP, but instead perform a BK, and UP, and a P, and then do the insertion. The corresponding also holds true for list commands.

Occasionally, the user will want to employ the S command in a macro to save some temporary result. For example, the SW command could be defined as:

```
(M (SW) (N M)
    (NTH N)
    (S FOO 1)
    MARK
    0
    (NTH M)
    (S FIE 1)
    (I 1 FOO)
    ←←
    (I 1 FIE))
```

Since this version of SW sets FOO and FIE, using SW may have undesirable side effects, especially when the editor was called from deep in a computation, we would have to be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The BIND command solves both problems.

(BIND $COMS_1 \cdots COMS_N$) [Editor Command]

Binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands $COMS_1 \cdots COMS_N$. Note that these bindings are only in effect while the commands are being executed, and that BIND can be used recursively;

it will rebind #1, #2, and #3 each time it is invoked.

BIND is implemented by (PROG (#1 #2 #3) (EDITCOMS (CDR *COM*))) where *COM* corresponds to the entire BIND command, and EDITCOMS is an internal editor function which executes a list of commands.

Thus we could now write SW safely as:

```
(M (SW) (N M)
    (BIND (NTH N)
          (S #1 1)
          MARK
          0
          (NTH M)
          (S #2 1)
          (I 1 #1)
          ←←
          (I 1 #2)))
```

(ORIGINAL *COMS*$_1$ ··· *COMS*$_N$) [Editor Command]

Executes *COMS*$_1$ ··· *COMS*$_N$ without regard to macro definitions. Useful for redefining a built in command in terms of itself., i.e. effectively allows user to "advise" edit commands.

User macros are stored on a list USERMACROS. The file package command USERMACROS (page 11.24), is available for dumping all or selected user macros.

17.17 UNDO

Each command that causes structure modification automatically adds an entry to the front of UNDOLST that contains the information required to restore all pointers that were changed by that command.

UNDO [Editor Command]

Undoes the last, i.e., most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., MBD UNDONE. The edit chain is then *exactly* what it was before the "undone" command had been performed. If there are no commands to undo, UNDO types NOTHING SAVED.

!UNDO [Editor Command]

Undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints NOTHING SAVED.

Undoing an event containing an I, E, or S command will also undo the side effects of the evaluation(s), e.g., undoing (I 3 (/NCONC FOO FIE)) will not only restore the 3rd element but also restore FOO. Similarly, undoing an S command will undo the set. See the discussion of UNDO in page 8.11. (Note that if the I command was typed directly to the editor, /NCONC would automatically be substituted for NCONC as described in page 8.22.)

Since UNDO and !UNDO cause structure modification, they also add an entry to UNDOLST. However, UNDO and !UNDO entries are skipped by UNDO, e.g., if the user performs an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, the user can also specify precisely which commands he wants undone by identifying the corresponding entry on the history list. In this case, he can undo an UNDO command, e.g., by typing UNDO UNDO, or undo a !UNDO command, or undo a command other than that most recently performed.

Whenever the user *continues* an editing session, the undo information of the previous session is protected by inserting a special blip, called an undo-block, on the front of UNDOLST. This undo-block will terminate the operation of a !UNDO, thereby confining its effect to the current session, and will similarly prevent an UNDO command from operating on commands executed in the previous session.

Thus, if the user enters the editor continuing a session, and immediately executes an UNDO or !UNDO, the editor will type BLOCKED instead of NOTHING SAVED. Similarly, if the user executes several commands and then undoes them all, another UNDO or !UNDO will also cause BLOCKED to be typed.

UNBLOCK [Editor Command]

Removes an undo-block. If executed at a non-blocked state, i.e., if UNDO or !UNDO *could* operate, types NOT BLOCKED.

TEST [Editor Command]

Adds an undo-block at the front of UNDOLST.

Note that TEST together with !UNDO provide a "tentative" mode for editing, i.e., the user can perform a number of changes, and then undo all of them with a single !UNDO command.

(UNDO *EventSpec*) [Editor Command]

EventSpec is an event specification (see page 8.5). Undoes the indicated event on the history list. In this case, the event does not have to be in the current editing session, even if the previous session has not been unblocked as described above. However, the user does have to be editing the same expression as was being edited in the indicated event.

If the expressions differ, the editor types the warning message "different expression", and does not undo the event. The editor enforces this to avoid the user accidentally undoing a random command by giving the wrong event specification.

17.18 EDITDEFAULT

Whenever a command is not recognized, i.e., is not "built in" or defined as a macro, the editor calls an internal function, EDITDEFAULT, to determine what action to take.[29] If a location specification is being

[29] Since EDITDEFAULT is part of the edit block, the user cannot advise or redefine it as a means of augmenting or extending the editor. However, the user can accomplish this via EDITUSERFN. If the value of the variable EDITUSERFN is T, EDITDEFAULT calls the function EDITUSERFN giving it the command as an argument. If EDITUSERFN returns a non-NIL value, its value is interpreted as a single command and executed. Otherwise, the error correction procedure described below is performed.

executed, an internal flag informs EDITDEFAULT to treat the command as though it had been preceded by an F.

If the command is a list, an attempt is made to perform spelling correction on CAR of the command[30] using EDITCOMSL, a list of all list edit commands.[31] If spelling correction is successful, the correct command name is RPLACAed into the command, and the editor continues by executing the command. In other words, if the user types (LP F PRINT (MBBD AND (NULL FLG))), only one spelling correction will be necessary to change MBBD to MBD. If spelling correction is not successful, an error is generated.

If the command is atomic, the procedure followed is a little more elaborate.

(1) If the command is one of the list commands, i.e.,. a member of EDITCOMSL, and there is additional input on the same terminal line, treat the entire line as a single list command.[32] Thus, the user may omit parentheses for any list command typed in at the top level (provided the command is not also an atomic command, e.g. NX, BK. For example,

```
*p
(COND (& &) (T &))
*XTR 3 2]
*MOVE TO AFTER LP
*
```

If the command is on the list EDITCOMSL but no additional input is on the terminal line, an error is generated, e.g.

```
*p
(COND (& &) (T &))
*MOVE

MOVE ?
*
```

If the command is on EDITCOMSL, and *not* typed in directly, e.g., it appears as one of the commands in a LP command, the procedure is similar, with the rest of the command stream at that level being treated as "the terminal line", e.g. (LP F (COND (T &)) XTR 2 2).[33]

(2) If the command was typed in and the first character in the command is an 8, treat the 8 as a mistyped left parenthesis, and and the rest of the line as the arguments to the command, e.g.,

```
*p
(COND (& &) (T &))
```

[30] unless DWIMFLG=NIL.

[31] When a macro is defined via the M command, the command name is added to EDITCOMSA or EDITCOMSL, depending on whether it is an atomic or list command. The USERMACROS file package command is aware of this, and provides for restoring EDITCOMSA and EDITCOMSL.

[32] The line is read using READLINE (page 8.30). Thus the line can be terminated by a square bracket, or by a carriage return not preceded by a space.

[33] Note that if the command is being executed in location context, EDITDEFAULT does not get this far, e.g., (MOVE TO AFTER COND XTR 3) will search for XTR, *not* execute it. However, (MOVE TO AFTER COND (XTR 3)) will work.

```
*8-2 (Y (RETURN Z)))
=(-2
*P
(COND (Y &) (& &) (T &))
```

(3) If the command was typed in, is the name of a function, and is followed by NIL or a list CAR of which is not an edit command, assume the user forgot to type E and means to apply the function to its arguments, type =E and the function name, and perform the indicated computation, e.g.

```
*BREAK(FOO)
=E BREAK
(FOO)
*
```

(4) If the last character in the command is P, and the first *N*-1 characters comprise a number, assume that the user intended two commands, e.g.,

```
*P
(COND (& &) (T &))
*0P
=0 P
(SETQ X (COND & &))
```

(5) Attempt spelling correction using EDITCOMSA, and if successful, execute the corrected command.

(6) If there is additional input on the same line, or command stream, spelling correct using EDITCOMSL as a spelling list, e.g.,

```
*MBBD SETQ X
=MBD
*
```

(6) Otherwise, generate an error.

17.19 EDITOR FUNCTIONS

(EDITF *NAME* COM_1 COM_2 ··· COM_N) [NLambda NoSpread Function]

Nlambda, nospread function for EDITing a Function. *NAME* is the name of the function, COM_1, COM_2, ···, COM_n are (optional) edit commands.

The value of EDITF is *NAME*.

The action of EDITF is somewhat complicated:

(1) In the most common case, if the definition of *NAME* is an EXPR (not as a result of its being broken or advised), and EDITF simply performs (PUTD *NAME* (EDITE (GETD '*NAME*) (LIST 'COM_1 'COM_2 ··· 'COM_N) '*NAME* 'FNS)).

(2) If *NAME* is an EXPR by virtue of its being broken or advised, and the original definition is also an EXPR, then the broken/advised definition is given to EDITE to be edited (since any changes there will also affect the original definition because all changes are destructive). However, a warning message is printed to alert the user that he must first position himself correctly before he can begin typing commands such as (-3 --), (N --), etc.

(3) If *NAME* is an EXPR by virtue of its being broken or advised, the original definition is not an EXPR, there is no EXPR property, and the file package "knows" which file *NAME* is contained in (see EDITLOADFNS?, page 17.58), then the EXPR definition of *NAME* is loaded onto its property list as described below, and the EDITF proceeds to the next possibility. Otherwise, a warning message is printed, and the edit proceeds, e.g., the user may have called the editor to examine the advice on a SUBR.

(4) If *NAME* is an EXPR by virtue of its being broken or advised, the original definition is not an EXPR, and there is an EXPR property, then the function is unbroken/unadvised (latter only with user's approval, since the user may really want to edit the advice) and EDITF proceeds to the next possibility.

(5) If *NAME* is not an EXPR, but has an EXPR property, EDITF prints PROP, and performs (EDITE (GETPROP '*NAME* 'EXPR) (LIST '*COM*$_1$ '*COM*$_2$ ··· '*COM*$_N$) '*NAME* 'PROP). In this case, if the edit completes and no changes have been made, EDITE prints NOT CHANGED, SO NOT UNSAVED. If changes were made, but the value of DFNFLG (page 5.9) is PROP, EDITE prints CHANGED, BUT NOT UNSAVED. Otherwise if changes were made, EDITE prints UNSAVED and does an UNSAVEDEF.

(6) If *NAME* is neither an EXPR nor has an EXPR property, and the file package "knows" which file *NAME* is contained in (see EDITLOADFNS?, page 17.58), the EXPR definition of *NAME* is automatically loaded (using LOADFNS) onto the EXPR property, and EDITE proceeds as described above.[34] In addition, if *NAME* is a member of a block, the user will be asked whether he wishes the rest of the functions in the block to be loaded at the same time.[35]

(7) If *NAME* is neither an EXPR nor has an EXPR property, but it does have a definition, EDITF generates an *NAME* NOT EDITABLE error.

(8) If *NAME* is neither defined, nor has an EXPR property, but its top level value is a list, EDITF assumes the user meant to call EDITV, prints =EDITV, calls EDITV and returns. Similarly, if *NAME* has a non-NIL property list, EDITF prints =EDITP, calls EDITP and returns.

[34] Because of the existence of the file map (see page 11.38), this operation is extremely fast, essentially requiring only the time to perform the READ to obtain the actual definition.

[35] The editor's behaviour in this case is controlled by the value of EDITLOADFNSFLG, which is a dotted pair of two flags. The CAR of EDITLOADFNSFLG controls the loading of the function, and the CDR controls the loading of the block. A value of NIL for either flag means "load but ask first," a value of T means "don't ask, just do it" and anything else means "don't ask, don't do it." The initial value of EDITLOADFNSFLG is (T . NIL), meaning to load the function without asking, and ask about loading the block.

(9) If *NAME* is neither a function, nor has an EXPR property, nor a top level value that is a list, nor a non-NIL property list, EDITF attempts spelling correction using the spelling list USERWORDS,[36] and, if successful, goes back to the beginning.

(10) Otherwise, EDITF generates an *NAME* NOT EDITABLE error.

In all cases, if a function is edited, and changes were made, the function is time-stamped (by EDITE), which consists of inserting a comment of the form (* *USERS-INITIALS DATE*) (see page 17.60). If the function was already time-stamped, then only the date is changed.

(EDITFNS *NAME* COM_1 COM_2 ··· COM_N) [NLambda NoSpread Function]

An nlambda, nospread function, used to perform the same editing operations on several functions. *NAME* is evaluated to obtain a list of functions.[37] COM_1, COM_2, ···, COM_N are (optional) edit commands. EDITFNS maps down the list of functions, prints the name of each function, and calls the editor (via EDITF) on that function. The value of EDITFNS is NIL.

For example, (EDITFNS FOOFNS (R FIE FUM)) will change every FIE to FUM in each of the functions on FOOFNS.

The call to the editor is ERRORSET protected, so that if the editing of one function causes an error, EDITFNS will proceed to the next function. In particular, if an error occurred while editing a function via its EXPR property, the function would not be unsaved. Thus in the above example, if one of the functions did not contain a FIE, the R command would cause an error, it would not be unsaved, and editing would continue with the next function.

(EDITV *NAME* COM_1 COM_2 ··· COM_N) [NLambda NoSpread Function]

Similar to EDITF, for editing values of variables.

The value of EDITV is the name of the variable whose value was edited.

If *NAME* is a list, it is evaluated and its value given to EDITE, e.g., (EDITV (CDR (ASSOC 'FOO DICTIONARY))). In this case, the value of EDITV is T.

However, for most applications, *NAME* is a variable name, i.e., atomic, as in EDITV(FOO). If the value of this variable is NOBIND, EDITV checks to see if it is the name of a function, and if so, assumes the user meant to call EDITF, prints =EDITF, calls EDITF and returns. Otherwise, EDITV attempts spelling correction using the list USERWORDS.[38] Then EDITV will call EDITE on the value of *NAME* (or the corrected spelling thereof), and *TYPE*=VARS. Thus, if the value of FOO is NIL, and the user performs (EDITV FOO), no spelling correction will occur, since FOO is the name of a variable in the user's system, i.e., it has a value. However, EDITE will generate an error, since FOO's value is not a list, and hence

[36] Unless DWIMFLG=NIL. Spelling correction is performed using the function MISSPELLED? (page 15.18). If *NAME*=NIL, MISSPELLED? returns the last "word" referenced, e.g., by DEFINEQ, EDITF, PRETTYPRINT etc. Thus if the user defines FOO and then types (EDITF), the editor will assume he meant FOO, type =FOO, and then type EDIT.

[37] If *NAME* is atomic, and its value is not a list, and it is the name of a file, (FILEFNSLST '*NAME*) will be used as the list of functions to be edited.

[38] Unless DWIMFLG=NIL. MISSPELLED? is also called if *NAME* is NIL, so that (EDITV) will edit LASTWORD.

not editable. If the user performs (EDITV FOOO), where the value of FOOO is NOBIND, and FOO is on the user's spelling list, the spelling corrector will correct FOOO to FOO. Then EDITE will be called on the value of FOO. Note that this may still result in an error if the value of FOO is not a list.

(EDITP *NAME* COM_1 COM_2 ... COM_N) [NLambda NoSpread Function]

Similar to EDITF for editing property lists. If the property list of *NAME* is NIL, EDITP attempts spelling correction using USERWORDS. Then EDITP calls EDITE on the property list of *NAME*, (or the corrected spelling thereof), with *TYPE*=PROPLST. When (if) EDITE returns, EDITP calls SETPROPLIST on *NAME* with the value returned.

The value of EDITP is the atom whose property list was edited.

(EDITE *EXPR COMS ATM TYPE IFCHANGEDFN*) [Function]

Edits the expression, *EXPR*, by calling EDITL on (LIST *EXPR*) and returning the last element of the value returned by EDITL. Generates an error if *EXPR* is not a list.

ATM and *TYPE* are for use in conjunction with the file package. If supplied, *ATM* is the *name* of the object that *EXPR* is associated with, and *TYPE* describes the association (i.e., *TYPE* corresponds to the *TYPE* argument of MARKASCHANGED, page 11.11.) For example, if *EXPR* is the definition of FOO, *ATM*=FOO and *TYPE*=FNS. When EDITE is called from EDITP, *EXPR* is the property list of *ATM*, and *TYPE*=PROPLST, etc..

EDITE calls EDITL to do the editing (described below). Upon return, if both *ATM* and *TYPE* are non-NIL, ADDSPELL is called to add *ATM* to the appropriate spelling list. Then, if *EXPR* was changed,[39] and the value of *IFCHANGEDFN* is not NIL, the value of *IFCHANGEDFN* is applied to the arguments *ATM*, *EXPR*, *TYPE*, and a flag which is T for normal edits from editor, NIL for calls that were aborted via control-D or STOP. Otherwise, if *EXPR* was changed, and the value of *IFCHANGEDFN* is NIL, and *TYPE* is not NIL, MARKASCHANGED (page 11.11) is called on *ATM* and *TYPE*. EDITE uses RESETSAVE to insure that *IFCHANGEDFN* and MARKASCHANGED are called if any change was made even if editing is subsequently aborted via control-D. (In this case, the fourth argument to *IFCHANGEDFN* wil be NIL.)

(EDITL *L COMS ATM MESS EDITCHANGES*) [Function]

EDITL *is* the editor. Its first argument is the edit chain, and its value is an edit chain, namely the value of *L* at the time EDITL is exited.[40]

COMS is an optional list of commands. For interactive editing, coms is NIL. In this case, EDITL types EDIT (or *MESS*, if it not NIL) and then waits for input from terminal. All input is done with EDITRDTBL as the readtable. Exit occurs only via an OK, STOP, or SAVE command.

[39] For *TYPE*=FNS or *TYPE*=PROP, i.e., calls from EDITF, EDITE performs some additional operations as described earlier under EDITF.

[40] *L* is a SPECVAR, and so can be examined or set by edit commands. For example, ↑ is equivalent to (E (SETQ L (LAST L)) T). However, the user should only manipulate or examine *L* directly as a last resort, and then with caution.

If *COMS* is *not* NIL, no message is typed, and each member of *COMS* is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed, the rest of the commands are ignored, and EDITL exits with an error, i.e., the effect is the same as though a STOP command had been executed. If all commands execute successfully, EDITL returns the current value of *L*.

ATM is optional. On calls from EDITF, it is the name of the function being edited; on calls from EDITV, the name of the variable, and calls from EDITP, the atom whose property list is being edited. The property list of *ATM* is used by the SAVE command for saving the state of the edit. Thus SAVE will not save anything if *ATM*=NIL, i.e., when editing arbitrary expressions via EDITE or EDITL directly.

EDITCHANGES is used for communicating with EDITE.

(EDITLO *L COMS MESS* —) [Function]

Like EDITL, except it does not rebind or initialize the editor's various state variables, such as LASTAIL, UNFIND, UNDOLST, MARKLST, etc. Should only be called when already under a call to EDITL.

(EDIT4E *PAT X* —) [Function]

The editor's pattern match routine. Returns T, if *PAT* matches *X*. See page 17.13 for definition of "match".

Note: Before each search operation in the editor begins, the entire pattern is scanned for atoms or strings containing $s (<esc>s). Atoms or strings containing $s are replaced by lists of the form ($ ···), and atoms or strings ending in double $s are replaced by lists of the form ($$ ···). Thus from the standpoint of EDIT4E, single and double $ patterns are detected by (CAR *PAT*) being the atom $ (<esc>) or the atom $$ (<esc><esc>). Therefore, if the user wishes to call EDIT4E directly, he must first convert any patterns which contain atoms or strings containing $s to the form recognized by EDIT4E. This is done with the function EDITFPAT:

(EDITFPAT *PAT* —) [Function]

Makes a copy of *PAT* with all atoms or strings containing $s (<esc>s) converted to the form expected by EDIT4E.

(EDITFINDP *X PAT FLG*) [Function]

Allows a program to use the edit find command as a pure predicate from outside the editor. *X* is an expression, *PAT* a pattern. The value of EDITFINDP is T if the command F *PAT* *would* succeed, NIL otherwise. EDITFINDP calls EDITFPAT to convert *PAT* to the form expected by EDIT4E, unless *FLG*=T. Thus, if the program is applying EDITFINDP to several different expressions using the same pattern, it will be more efficient to call EDITFPAT once, and then call EDITFINDP with the converted pattern and *FLG*=T.

(ESUBST *NEW OLD EXPR ERRORFLG CHARFLG*) [Function]

Equivalent to performing (R *OLD NEW*) with *EXPR* as the current expression, i.e., the order of arguments is the same as for SUBST. Note that *OLD* and/or *NEW* can employ $s (<esc>s). The value of ESUBST is the modified *EXPR*. Generates an error if *OLD* not found in *EXPR*. If *ERRORFLG*=T, also prints an error message of the form *OLD* ?.

If *CHARFLG*=T and no $s (<esc>s) are specified in *NEW* or *OLD*, it is equivalent to (RC *OLD* *NEW*). In other words, if *CHARFLG*=T, and no $s appear, ESUBST will supply them.

ESUBST is always undoable.

(EDITLOADFNS? *FN* *STR* *ASKFLG* *FILES*) [Function]

FN is the name of a function. EDITLOADFNS? returns the name of file *FN* is contained in, or NIL.

EDITLOADFNS? performs (WHEREIS *FN* FNS *FILES*) to obtain the name of the file(s) containing *FN*, if any (see page 11.10). If there is more than one file, EDITLOADFNS? asks the user to indicate which file. It then checks the FILEDATES property for each file to see if the version that was originally loaded still exists.[41] If the file that was *originally* loaded no longer exists, but there is a different version of the file on that directory, EDITLOADFNS? prints "****can't find *FILENAME*", and then uses the version that it could find. Similarly, if the original version *is* found, but a newer version is also found, EDITLOADFNS? prints "****Note: *FILENAME* is not the newest version" and then uses the newest version.

Having decided which file the function is on, if *ASKFLG*=NIL, EDITLOADFNS? prints the value of *STR* followed by the name of the file, and returns the name of the file. If *ASKFLG*=T, EDITLOADFNS? calls ASKUSER giving (LIST *FN* *STR* *FILENAME*) as *MESS*, the message to be printed. If ASKUSER returns Y, EDITLOADFNS? returns the filename. If *STR*=NIL, "loading from" is used.

EDITLOADFNS? is used by the editor, LOADFNS (when the file name is not supplied), by PRETTYPRINT, and by DWIM.

(CHANGENAME *FN* *FROM* *TO*) [Function]

Replaces all occurrences of *FROM* by *TO* in the definition of *FN*. If *FN* is an EXPR, CHANGENAME performs (NLSETQ (ESUBST *TO* *FROM* (GETD *FN*))). If *FN* is *compiled*, CHANGENAME searches the literals of *FN* (and all of its compiler generated subfunctions), replacing each occurrence of *FROM* with *TO*. This will succeed even if *FROM* is called from *FN* via a linked call. In this case, the call will also be relinked to call *TO* instead.

The value of CHANGENAME is *FN* if at least one instance of *FROM* was found, otherwise NIL.

CHANGENAME is used by BREAK and ADVISE for changing calls to FN_1 to calls to FN_1-IN-FN_2.

The function EDITCALLERS provides a way of rapidly searching a file or entire set of files, even files not loaded into Interlisp or "noticed" by the file package, for the appearance of one or more key words (atoms) anywhere in the file.

[41] In the case that *FILES*=T and the WHEREIS package has been loaded (page 23.40), files(s) may be found that have not been loaded or otherwise noticed, and thus will not have FILEDATES property. In this case, EDITLOADFNS? does not do any version checks, but simply uses the latest version.

(EDITCALLERS *ATOMS FILES COMS*) [Function]

Uses FFILEPOS to search the file(s) *FILES* for occurrences of the atom(s) *ATOMS*. It then calls EDITE on each of those objects,[42] performing the edit commands *COMS*. If *COMS*=NIL, then (EXAM . *ATOMS*) is used. Both *ATOMS* and *FILES* may be single atoms. If *FILES* is NIL, FILELST is used. Elements on *ATOMS* may contain $s (<esc>s).

EDITCALLERS prints the name of each file as it searches it, and when it finds an occurrence of one of *ATOMS*, it prints out either the name of the containing function or, if the atom occurred outside a function definition, it prints out the byte position that the atom was found.

EDITCALLERS will read in and use the filemap of the file. In the case that the editor is actually called, EDITCALLERS will LOADFROM the file if the file has not previously been noticed.

(FINDCALLERS *ATOMS FILES*) [Function]

Like EDITCALLERS, except does not call the editor, but instead simply returns the list of files that contain one of *ATOMS*.

(EDITRACEFN *COM*) [Function]

Is available to help the user debug complex edit macros, or subroutine calls to the editor. If EDITRACEFN is set to T, the function EDITRACEFN is called whenever a command that was not typed in by the user is about to be executed, giving it that command as its argument. However, the TRACE and BREAK options described below are probably sufficient for most applications.

If EDITRACEFN is set to TRACE, the name of the command and the current expression are printed. If EDITRACEFN=BREAK, the same information is printed, and the editor goes into a break. The user can then examine the state of the editor.

EDITRACEFN is initially NIL.

(SETTERMCHARS *NEXTCHAR BKCHAR LASTCHAR UNQUOTECHAR 2CHAR PPCHAR*) [Function]

Used to set up the immediate read macros used by the editor, as well as the control-Y read macro (page 6.39). *NEXTCHAR*, *BKCHAR*, *LASTCHAR*, *2CHAR* and *PPCHAR* specify which control character should perform the edit commands NXP, BKP, -1P, 2P and PP*, respectively; *UNQUOTECHAR* corresponds to control-Y. For each non-NIL argument, SETTERMCHARS makes the corresponding control character have the indicated function. The arguments to SETTERMCHARS can be character codes, the control characters themselves, or the alphabetic letters corresponding to the control characters.

If an argument to SETTERMCHARS is currently assigned as an interrupt character, it cannot be a read macro (since the reader will never see it); SETTERMCHARS prints a message to that effect and makes no change to the control character. However, if SETTERMCHARS is given a list as one of its arguments, it uses CAR of the list even if the character is an interrupt. In this case, if CADR of the list is non-NIL, SETTERMCHARS reassigns the interrupt function to CADR. For example, if control-X is an interrupt,

[42] EDITCALLERS uses GETDEF (page 11.17) to obtain the "definition" for each object. When EDITE returns, if a change was made, PUTDEF is called to store the changed object.

(SETTERMCHARS '(X W)) assigns control-W the interrupt control-X had, and makes control-X be the *NEXTCHAR* operator.

As part of the greeting operation, SETTERMCHARS is applied to the value of EDITCHARACTERS, which is initially (J X Z Y N) in Interlisp-D and in Interlisp-10 under Tenex, (J A L Y K) under Tops-20 (control-J is line-feed). SETTERMCHARS is called *after* the user's init file is loaded, so it works to reset EDITCHARACTERS in the init file; alternatively, SETTERMCHARS can be called explicitly.

17.20 TIME STAMPS

Whenever a function is edited, and changes were made, the function is time-stamped (by EDITE), which consists of inserting a comment of the form (* *USERS-INITIALS DATE*). *USERS-INITIALS* is the value of the variable INITIALS. After greeting, or following a SYSIN, the function SETINITIALS is called. SETINITIALS searches INITIALSLST, a list of elements of the form (*USERNAME* . *INITIALS*) or (*USERNAME FIRSTNAME INITIALS*). If the user's name is found, INITIALS is set accordingly. If the user's name is *not* found on INITIALSLST, INITIALS is set to the value of DEFAULTINITIALS, initially edited:. Thus, the default is to always time stamp. To suppress time stamping, the user must either include an entry of the form (*USERNAME*) on INITIALSLST, or set DEFAULTINITIALS to NIL before greeting, i.e. in his user profile, or else, *after* greeting, explicitly set INITIALS to NIL.

If the user wishes his functions to be time stamped with his initials when edited, he should include a file package command command of the form (ADDVARS (INITIALSLST (*USERNAME* . *INITIALS*))) in the user's INIT.LISP file (see page 14.5).

The following three functions may be of use for specialized applications with respect to time-stamping: (FIXEDITDATE *EXPR*) which, given a lambda expression, inserts or smashes a time-stamp comment; (EDITDATE? *COMMENT*) which returns T if *COMMENT* is a time stamp; and (EDITDATE *OLDATE INITLS*) which returns a new time-stamp comment. If *OLDATE* is a time-stamp comment, it will be reused.

CHAPTER 18

INTERLISP-D SPECIFICS

Interlisp-D is an implementation of the Interlisp language that runs on the Xerox 1100, 1108, and 1132 machines. It is completely upward compatable with the older Interlisp-10, except as specified in this manual. The most significant extension to Interlisp is the window display package, described on page 19.1. However, Interlisp-D also offers many other extensions, which are described in detail below.

18.1 INTERLISP-D INTERRUPT CHARACTERS

The table below gives the interrupt characters currently enabled in Interlisp-D. Many of these are the same as those used in the Tenex version of Interlisp-10, but some have been removed, and some have had their meanings changed. It is possible to change the assignments of control characters to interrupts using INTERRUPTCHAR (page 9.17).

Note: In Interlisp-D with multiple processes, it is not sufficient to say that "the computation" is broken, aborted, etc; it is necessary to specify which process is being acted upon. Most of the interrupt characters below refer to the TTY process, which is the one currently receiving keyboard input. Control-H can be used to break arbitrary processes. For more information, see page 18.35.

control-B — Causes a break within the TTY process. Use control-H to break a particular process. Note that this break occurs at the next function call, so it is like control-H in Interlisp-10; it is always safe to resume the computation. There is no interrupt character like control-B in Interlisp-10

control-C — On the Xerox 1100 and Xerox 1132, brings the user into the Raid low-level debugger. From Raid, typing control-N resumes the Lisp computation, and control-D resets the stack. On the Xerox 1108, after typing control-C, the system stops and waits for the next character typed. Pressing the STOP key will do a HARDRESET, returning control to the user. Pressing the UNDO key will start up the TeleRaid debugger.

control-D — Aborts the TTY process, and unwinds its stack to the top level. Calls RESET (page 9.14).

control-E — Aborts the TTY process, and unwinds its stack to the last ERRORSET. Calls ERROR! (page 9.14).

control-H — Pops up a menu listing all of the currently-running processes. Selecting one of the processes will cause the break to take place in that process.

control-P — Changes the PRINTLEVEL setting, as described on page 6.18.

control-T — Prints status information for the TTY process.

Note: The control-O, and control-S interrupt characters from the Tenex version of Interlisp-10 are not enabled in Interlisp-D.

18.2 GARBAGE COLLECTION

Interlisp-D has a reference-counting garbage collector (Interlisp-10 uses the more familiar mark-and-sweep algorithm). A reference-counting garbage collector uses time proportional to the garbage being collected and not to the size of the address space. This is a crucial advantage for a large address space system such as Interlisp-D. It does have a disadvantage in that circular lists are never reclaimed, as their reference count never goes to zero. In addition, atoms are currently not garbage collected; and non-atomic hash array keys are not collected (in Interlisp-10, when a non-atomic hash key is no longer referenced except by the hash array itself, the hashlink goes away and both the key and the value, if it is nowhere else referenced, are reclaimed).

Garbage collection in Interlisp-D is controlled by the following functions and variables:

(RECLAIM) [Function]

Initiates a garbage collection. RECLAIM always returns 0, independent of the actual number of cells collected.

(RECLAIMMIN *N*) [Function]

The frequency of garbage collection is user settable via the function RECLAIMMIN (which plays a role similar to Interlisp-10's MINFS, which is a no-op in Interlisp-D). Lisp keeps track of the number of cells of any type that have been allocated; when it reaches the RECLAIMMIN number, a garbage collection occurs. (RECLAIMMIN *N*) returns the current setting of the parameter, and, if *N* is non-NIL, sets it to *N*.

As there is no motivation for the Interlisp-10 CTRL-S interrupt, it is not enabled.

RECLAIMWAIT [Variable]

Interlisp-D will invoke a RECLAIM if the system is idle and waiting for user input for RECLAIMWAIT seconds (currently set for 4 seconds).

(GCGAG *MESSAGE*) [Function]

GCGAG sets the message that appears on the display screen while a garbage collection is taking place. If *MESSAGE* is non-NIL, the cursor is complemented during a RECLAIM; if *MESSAGE*=NIL, nothing happens. This limited choice exists because it was found that printing a message took a significant fraction of the time of small RECLAIMs. The value of GCGAG is its previous setting.

(GCTRP) [Function]

The function GCTRP returns the number of cells (of any type, not just LISTP) until the next garbage collection, according to the RECLAIMMIN number, although this number is not very meaningful.

18.3 VARIABLE BINDINGS

Interlisp-D uses deep binding of variables, whereas Interlisp-10 currently uses shallow binding (prior to 1975, Interlisp-10 used deep binding). Although this makes little difference for most programs, it can make a difference in efficiency of execution. For example, it is better to pass parameters as arguments than to let subfunctions reference them freely. In addition, declaring variables that are never bound (i.e., whose top level value only is used) to be GLOBALVARS is important. Sloppy Interlisp-10 code that rebinds variables that have been declared as GLOBALVARS will not run correctly in Interlisp-D. Be careful to use RESETVARS to "rebind" variables that are declared GLOBALVARS. RESETVARS works in both systems; in a shallow system, RESETVARS just binds its arguments as PROG variables (and makes sure they are declared SPECVARS), while in a deep system such as Interlisp-D, entries are made on RESETVARSLST. If the compiler sees an attempt to bind a global variable, it will print out an error message.

For performance reasons, it is important to declare global variables as such in Interlisp-D. This can be done with the GLOBALVARS file package command (page 11.25), which causes variables to be declared as global to the compiler. For more information on variable bindings and performance, see page 18.19.

18.4 STACK FORMAT

Both the interpreter and compiler generate different intermediate frames than are found in Interlisp-10, so if the user has code that assumes a particular number of frames will exist at some point (e.g., using STKNTH), it will probably be wrong. STKPOS and STKSCAN are still available, however, and REALSTKNTH and REALFRAMEP are useful for ignoring those intermediate frames.

18.5 SAVING VIRTUAL MEMORY STATE

The Interlisp-D virtual memory is kept in the file Lisp.virtualmem. As virtual memory pages are accessed, they are loaded from this file into real memory. To exit from Interlisp-D to the Alto Executive so that it is possible to return to the current Interlisp-D environment, it is necessary to save the state of the virtual memory. The simplest way is to use the function LOGOUT (page 14.2). This will write out all altered pages from real memory to Lisp.virtualmem.

If you are the sole user of Interlisp-D on a disk partition, then you will probably want to use LOGOUT. However, if other Interlisp-D users may be using that partition, and you wish to save your state, then it may be more appropriate to use SYSOUT (page 14.3). Note that SYSOUT in Interlisp-D saves the *entire* state of the virtual memory, instead of just the saved pages, so Interlisp-D sysout file are very large.

(VMEMSIZE) [Function]

Returns the number of pages in use in the virtual memory. This is the roughly the same as the number of pages required to make a sysout file on the local disk.

Interlisp-D contains a routine that writes out dirty pages of the virtual memory during I/O wait, assuming that swapping has caused at least one dirty page to be written back into Lisp.virtualmem (making it non-continuable). The frequency with which this routine runs is determined (inversely) by:

BACKGROUNDPAGEFREQ [Variable]

This global variable determines how often the routine that writes out dirty pages is run. Initially it is set to 4, so the dirty page routine is run once every 4 times around the idle loop. (The lower BACKGROUNDPAGEFREQ is set, the less responsiveness you get at typein, so it may not be desirable to set it all the way down to 1.)

The following function is used to write all of the dirty pages out, to make sure that the current state is not lost if there is a system crash.

(SAVEVM —) [Function]

This function is similar to logging out and continuing, but faster. It takes about as long as a logout, which can be as brief as 10 seconds or so if you have already written out most of your dirty pages by virtue of being idle a while. After the SAVEVM, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the SAVEVM) should there be a system crash or other disaster.

If the system has been idle long enough, dirty pages have been written, and there are few enough dirty pages left to write that a SAVEVM would be quick, SAVEVM will be automatically called. While SAVEVM is being executed, the cursor is changed to a special "SAV/ING" cursor. You can control how often SAVEVM is automatically called by setting the following two global variables:

SAVEVMWAIT [Variable]
SAVEVMMAX [Variable]

The system will call SAVEVM after being idle for SAVEVMWAIT seconds (initially 60) if there are fewer than SAVEVMMAX pages dirty (initially 600). These values are fairly conservative. If you want to be extremely wary, you can set SAVEVMWAIT=0 and SAVEVMMAX=10000, in which case SAVEVM will be called the first chance available after the first dirty page has been written.

18.6 ERROR TYPES

The following additional error types occur in Interlisp-D:

5	FILE SYSTEM ERROR
48	FLOATING UNDERFLOW
49	FLOATING OVERFLOW
50	OVERFLOW
51	ARG NOT HARRAY
52	TOO MANY ARGUMENTS

Interlisp-D allows the user to trap arithmetic exceptions. The action taken when overflow occurs may be set with the function OVERFLOW (page 2.38).

READ-MACRO CONTEXT errors are not generated in Interlisp-D. In the situation where Interlisp-10 would generate the error, the call to READ within the macro will simply return NIL.

18.7 COMPILER

Interlisp-D runs a different instruction set than Interlisp-10, so source files from Interlisp-10 must be recompiled. The default extension (value of COMPILE.EXT) for Interlisp-D compiled files is "DCOM" rather than "COM" as in Interlisp-10.

The Interlisp-10 compiler translates Lisp source programs into 36-bit PDP-10 instructions. The Interlisp-D compiler compiles Lisp source programs into an 8-bit Lisp instruction set executed by the Xerox 1100 family machines.

In Interlisp-D, block compiling is handled somewhat differently than in Interlisp-10; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance advantage. Block compiling in Interlisp-D works by automatically renaming the block functions with special names, and calling these functions with the normal function-calling mechanisms. Specifically, a function *FN* is renamed to \BLOCK-NAME/*FN*. For example, function FOO in block BAR is renamed to "\BAR/FOO". Note that it is possible with this scheme to break functions internal to a block.

Interlisp-D has an optimizing compiler. Among other optimizations, it performs constant folding. Variables can be declared by the user to be compiler constants using the file package command CONSTANTS (page 11.27), which is syntactically the same as VARS, but additionally informs the compiler that the "variables" are constants.

18.8 LINKED FUNCTION CALLS

Linked function calls are not implemented in Interlisp-D. One noticeable result of this is that if you break a function that is used by the system, for example in the READ-EVAL-PRINT loop, you will get unexpected breaks within system code. These extra breaks can be safely exited with OK. To avoid this inconvenience, BREAK the function inside another function, e.g., (BREAK (PRIN1 IN FOO)). (Note: Functions that begin with a backslash (\) are system internal functions and should not be broken or advised.)

18.9 HELPSYS

There is currently no HELPSYS facility in Interlisp-D. There are plans to reimplement a HELPSYS facility eventually.

18.10 OPERATING SYSTEM DEPENDENT FUNCTIONS

Many Interlisp-10 functions are missing from Interlisp-D. An attempt has been made to provide an appropriate implementation for the more useful of these functions, but some simply do not make sense on the Xerox 1100 family machines. For example, there is no such thing as a JSYS. Any function containing a call on JSYS or ASSEMBLE will fail to compile.

The following Interlisp-10 functions are not implemented in Interlisp-D: LISPXSTATS, SUBSYS, GETBLK, RELBLK, ERSTR, GTJFN, OPNJFN, RLJFN, OPENF, JFNS.

The following Interlisp-10 functions are implemented as dummies in Interlisp-D: LISPXWATCH, ADDSTATS, HOSTNA USERNUMBER, HOSTNUMBER, LOADAV. There are communication network analogs of HOSTNAME and HOSTNUMBER called ETHERHOSTNAME and ETHERHOSTNUMBER (page 21.5).

Additional Functions:

(HOSTNAMEP *NAME*) [Function]

Returns T if *NAME* is recognized as a valid device or remote file server name at the moment HOSTNAMEP is called.

(DIRECTORYNAMEP *DIRNAME HOSTNAME*) [Function]

Returns T if *DIRNAME* is recognized as a valid directory. *DIRNAME* may include an explicit hostname. If *HOSTNAME* is supplied, it is used instead. The connected directory and hostname are used as defaults.

(MACHINETYPE) [Function]

Returns the type of machine that Interlisp-D is running on: either DORADO (for the Xerox 1132), DOLPHIN (for the Xerox 1100), or DANDELION (for the Xerox 1108).

(RINGBELLS) [Function]

On the Xerox 1100, this flashes (reverse-videos) the screen several times. On the Xerox 1108, this also beeps through the keyboard speaker.

18.11 IDATE FORMAT

Interlisp-D uses a different time standard than Tenex does. IDATE still has the essential property that (IDATE *X*) is less than (IDATE *Y*) if *X* is before *Y*, and (IDATE (GDATE *N*)) equals *N*. If the particular internal format of the integer date is being used to do arithmetic on dates, the user's programs must be fixed. But in that case the user is already in trouble with Interlisp-10, where the date standard is subtly different between Tenex and Tops20. The most useful property that the three formats have in common is that an internal date can be incremented by an integral number of days by computing as the "1 day" constant (which can be evaluated at compile time) the difference between two convenient IDATE's, e.g. (IDIFFERENCE (IDATE " 2-JAN-80 12:00") (IDATE " 1-JAN-80 12:00")).

Currently, the format argument of DATE and GDATE is not supported (an error will occur if the user tries to give one). IDATE now parses most of the date forms allowed in Interlisp-10; e.g., the month can be given numerically, slashes can be used as separators, extra spaces are ignored.

(SETTIME *DATE&TIME*) [Function]

Sets the internal time-of-day clock. If *DATE&TIME* = NIL, SETTIME attempts to get the time from the communications net; if it fails, the user is prompted for the time. If *DATE&TIME* is a string in a form that IDATE recognizes, it is used to set the time.

\TimeZoneComp [Variable]

This variable should be initialized (in {DSK}INIT.LISP) to the time-zone compensation, i.e., the number of hours west of GMT. For the U.S. west coast it is 8. For the east coast it is 5.

18.12 CHARACTER SET

Interlisp-D uses an 8-bit character set whereas Interlisp-10 uses standard 7-bit ASCII. The values returned by CHCON1 range from 0 to 255, and codes in this range are acceptable arguments to CHARACTER and FCHARACTER. Characters 0-127 have their standard ASCII interpretations; characters 128-255 are called "meta" characters. Some of the meta characters have printed representations in some fonts (for accents, ligatures, etc.), but most of them will be invisible if printed directly to the screen. Accordingly, the echoing conventions normally defined for control characters have been extended to apply also to meta characters. The echomode of any character may be set by the new function ECHOCHAR (page 6.43). In the original terminal table, the INDICATE character mode is specified for all meta characters, so all meta characters are echoed as a cross-hatch (#) followed by the printed representation corresponding to the 7 rightmost bits of the character. For example, character 129 is echoed as #↑A. There is currently no type-in syntax for meta characters.

The CHARCODE function (page 2.12), defined in both Interlisp-D and Interlisp-10, can be useful when dealing with the Interlisp-D character set.

18.13 READ TABLES

In Interlisp-D, all control characters are defined as separator characters in FILERDTBL, so that the font information in files is ignored when files are loaded. Users who run in both Interlisp-10 and Interlisp-D with the same files will want to make the same setting in Interlisp-10's FILERDTBL, in order that files created in one system can be read in the other. The appropriate expression to evaluate, which may be in your Interlisp-10 INIT.LISP file, is:

```
(SETSEPR '(1 2 3 4 5 6 7 9 10 11 12 13 14 15
           16 17 18 19 20 21 22 23 24 25 26)
         1 FILERDTBL)
```

18.14 KEYBOARD INTERPRETATION

In Interlisp-D, keyboard and mouse interpretation is now done entirely by Lisp code, and certain lower level keyboard facilities are therefore available. For each key on the keyboard/mouse there is a corresponding bit in memory that the hardware/microcode turn on and off as the key moves up and down. System-level routines decode the meaning of key transitions according to a table of "key actions", which may be to put particular Ascii codes in the sysbuffer, cause interrupts, change the internal shift/control status, or create events to be placed in the mouse buffer.

`(KEYDOWNP` *KEYNAME*`)` [Function]

Used to read the instantaneous state of any key, independent of any buffering or pre-assigned key action. Returns `T` if the key named *KEYNAME* is down at the moment the function is executed. Most keys are named by any of the characters on the key-top. The shift keys are named separately as `RSHIFT` and `LSHIFT`, space is `SPACE`, the unmarked keys are `BLANK-TOP`, `BLANK-MIDDLE`, and `BLANK-BOTTOM`, and the mouse buttons are `LEFT`, `MIDDLE`, and `RIGHT`. Paddles on the keyset (not generally available) are named `PAD1` through `PAD5`. Thus `(KEYDOWNP 'a)` returns `T` if the "a" key is down, `(KEYDOWNP 'TAB)` returns the state of the `TAB` key, etc.

`(KEYACTION` *KEYNAME ACTIONS*`)` [Function]

Changes the internal tables that define the action to be taken when a key transition is detected by the system keyboard handler. *KEYNAME* is specified as for `KEYDOWNP`. *ACTIONS* is a dotted pair of the form `(`*DOWN-ACTION* `.` *UP-ACTION*`)`, where the acceptable transition actions and their interpretations are:

`NIL` Take no action on this transition (the default for up-transitions on all ordinary characters).

a list `(`*CHAR SHIFTEDCHAR LOCKFLAG*`)`

CHAR and *SHIFTEDCHAR* are either ascii codes or non-digit characters standing for their ascii codes. When the transition occurs, *CHAR* or *SHIFTEDCHAR* is transmitted to the system buffer, depending on whether either of the 2 shift keys are down. *LOCKFLAG* is optional, and may be `LOCKSHIFT` or `NOLOCKSHIFT`. If *LOCKFLAG* is `LOCKSHIFT`, then *SHIFTEDCHAR* will also be transmitted when the `LOCK` shift is down (the alphabetic keys initially specify `LOCKSHIFT`, but the digit keys specify `NOLOCKSHIFT`).

Examples: `(a A LOCKSHIFT)` and `(61Q ! NOLOCKSHIFT)` are the initial settings for the down transitions of the "a" and "1" keys respectively.

`1SHIFTUP`, `2SHIFTUP`, `LOCKUP`, `CTRLUP`, `METAUP`
`1SHIFTDOWN`, `2SHIFTDOWN`, `LOCKDOWN`, `CTRLDOWN`, `METADOWN`

Change the status of the internal "shift" flags for the left shift, right shift, shift lock, ctrl, and meta keys, respectively. These shifts affect the interpretation of ordinary key actions. If either of the shifts is down, then *SHIFTEDCHAR*s are transmitted. If the lock flag is down, then *SHIFTEDCHAR*s are transmitted if the key action specified `LOCKSHIFT`. If the control flag is on, then the low-order five bits are masked out of the code that weuld otherwise be transmitted to the system buffer. If the meta flag is down, the high order (8th bit) is turned on as characters are transmitted.

Example: the initial *ACTIONS* for the left shift key is (1SHIFTUP . 1SHIFTDOWN).

EVENT An encoding of the current state of the mouse and selected keys is placed in the mouse-event buffer when this transition is detected.

KEYACTION returns the previous setting for *KEYNAME*. If *ACTIONS* is NIL, returns the previous setting without changing the tables.

(MODIFY.KEYACTIONS *KEYACTIONS SAVECURRENT?*) [Function]

KEYACTIONS is a list of key actions to be set, each of the form (*KEYNAME* . *ACTIONS*). The effect of MODIFY.KEYACTIONS is as if (KEYACTION *KEYNAME ACTIONS*) were performed for each item on *KEYACTIONS*.

If *SAVECURRENT?* is non-NIL, then MODIFY.KEYACTIONS returns a list of all the results from KEYACTION, otherwise it returns NIL. This can be used with a MODIFY.KEYACTIONS that appears in a RESETFORM, so that the list is built at "entry", but not upon "exit".

(METASHIFT *FLG*) [NoSpread Function]

If *FLG* is non-NIL, changes the keyboard handler (via KEYACTION) so as to interpret the bottom blank key ("swat") as a metashift: if a key is struck while meta is down, it is read with the 200Q bit set. For CHAT users this is a way of getting an "Edit" key on your simulated Datamedia. Returns previous setting.

18.15 LISPUSERS PACKAGES

Most of the LISPUSERS packages (see page 23.1) are available with the Interlisp-D system as separate loadable packages. The major exception is the HASH package, which is highly machine dependent, and the WHEREIS package which depends on it. EDITA, CJSYS, and many parts of the EXEC package are system-dependent by their very nature, and also are not included. The various network packages are not provided because many of these facilities are integrated into Interlisp-D at a more fundamental level.

Several packages not documented in the Interlisp Reference Manual are available. The list currently includes the following:

GRAPHER A collection of functions for laying out, displaying, and editing graphs on the Interlisp-D screen.

BROWSER Modifies the SHOW PATHS command of Masterscope so that the command's output is displayed as an undirected graph. Uses the GRAPHER package.

EVALSERVER Provides a set of routines to facilitate communication, over an Ethernet, between two or more Xerox 1100s running Interlisp-D.

HISTMENU Provides a simple way to access the Interlisp history list using a menu.

SAMEDIR This package advises MAKEFILE to notify the user if it appears that a file is being written onto a directory other than the one it came from, allowing the user to halt the process.

18.16 FILE SYSTEM

Typically, the most machine-dependent part of any computer language implementation is the I/O system. Regardless of efforts to create consistant interfaces, the fact remains that different physical machines offer different disks, printers, etc., and languages have to be extended to take advantage of these. In the case of implementing Interlisp on the Xerox 1100 family machines, the biggest change was the addition of facilities for using the high-resolution display, described elsewhere. Other changes have had to be made to accomodate using files on a local disk or on a file server, and sending files to remote printers. Every effort has been made to keep these interfaces compatible with Interlisp-10 conventions, to reduce the amount of work necessary when transferring programs. However, in some situations the user may wish to take advantage of the special extensions offered by Interlisp-D.

This section contains information about a variety of extensions to Interlisp-D that accomodate the different I/O environment.

18.16.1 File Names

"Full" file names inside of Interlisp-D look just like Tenex file names, except that all full file names begin with a device/host name (in braces) to identify the machine (or pseudo-machine) on which the file resides. Files on the local disk belong to device/host DSK, e.g. {DSK}FOO.BAR;3. PACKFILENAME and UNPACKFILENAME are still the appropriate way for programs to manipulate filenames. The device/host of a file may be accessed using the new field name HOST.

On Xerox 1100s and Xerox 1132s, Interlisp-D can access partitions other than the one which was booted. If the other partition is password-protected, Interlisp insists on the correct password before accessing any files. Partitions are denoted by {DSK1} for Partition 1, {DSK2} for Partition 2, etc. DIR, DIRECTORY, etc. all work for other partitions. Currently, SYSOUT does not work for partitions other than the default.

18.16.2 Renaming Files

Interlisp-D implements (RENAMEFILE *OLD NEW*) merely by copying *OLD* to *NEW* and then deleting *OLD*. While this is quite general (and even allows one to rename files from the local disk or one file server to another), it is slower than the Interlisp-10 RENAMEFILE operation. It also, in the case of renaming a local disk file, requires that the local disk have enough room to hold the copy of the file.

18.16.3 End Of Line Convention

Interlisp-D uses a different representation for *end of line* both internally and on files. Internally, *end of line* is represented by the carriage return character (15Q), whereas the internal representation in Interlisp-10 is the EOL character (37Q). The CHARCODE macro (page 2.12) is the appropriate way to code programs to be independent of the EOL convention: in all systems (CHARCODE EOL) is always the appropriate end-of-line character. (CHARCODE CR) and (CHARCODE TENEXEOL) provide the system-dependent character codes. Interlisp-D also interprets a carriage return/line feed sequence in a file as an end-of-line and reads it as a carriage return. TERPRI generates two characters in Interlisp-10, but only one in Interlisp-D.

18.16.4 Using Files with Processes

Currently, Interlisp-D does not provide interlocks to keep multiple processes from trying to access the same file. Therefore, the user has to be careful not to have two processes manipulating the same file at the same time. For example, it will not work to have one process TCOMPL a file while another process is running LISTFILES on it.

18.16.5 Miscellaneous File Manipulation

(COPYFILE *FROMFILE TOFILE*) [Function]

Copies a file to a new file. The source and destination may be any servers/devices. COPYFILE attempts to preserve the TYPE and CREATIONDATE where possible.

(DISKFREEPAGES — —) [Function]

Returns an estimate of the number of pages free on the local disk (current partition). This number is only a "hint", but is usually quite accurate.

(DISKPARTITION) [Function]

Returns the number of the current partition (1 or 2 on Xerox 1100, 1-5 on Xerox 1132).

18.16.6 Connecting to Directories

As in Interlisp-10, Interlisp-D has a notion of a "connected" directory, which is used as the default when you give a filename lacking an explicit device/host (and directory). The default is changed by using the programmer's assistant command CONN.

CONN {*DEVICE/HOST*}<*DIRECTORY*> [Prog. Asst. Command]

Either part of the argument is optional; if the directory is omitted, the default for devices that have directories is the value of (USERNAME); if the host is omitted, connection will be made to another directory on the same host as before. If CONN is given with no arguments, connects to the value of LOGINHOST/DIR.

Note that CONN does not require or provide any directory access privileges, as does the command of the same name in Interlisp-10. Access privileges are checked when a file is opened.

(CNDIR *HOST/DIR*) [Function]

Programmatic form of CONN. Connects to the directory *HOST/DIR*. Returns the fullname of the now-connected directory.

(/CNDIR *HOST/DIR*) [Function]

Undoable form of CNDIR. CONN is implemented via /CNDIR.

LOGINHOST/DIR [Variable]

CONN with no argument connects to the value of the variable LOGINHOST/DIR, initially {DSK}, but usually reset in the user's greeting file.

(DIRECTORYNAME *FLG STRPTR*) [Function]

Similar to Interlisp-10 USERNAME. If *FLG* is T, returns the currently connected host and directory name. If *FLG* is NIL, returns the value of LOGINHOST/DIR. If *STRPTR* is T, the value is returned as an atom, otherwise it is returned as a string.

DIRECTORIES [Variable]

Global variable containing the list of directories searched (in order) by SPELLFILE and FINDFILE (page 15.20) when not given an explicit *DIRLST* argument. In this list, the atom NIL stands for the login directory (LOGINHOST/DIR), and the atom T stands for the currently connected directory.

18.16.7 Binary I/O

Interlisp-D supports a datatype called a STREAM, whose basic operations are "input" and "output". They provide an efficient handle to an open file. All I/O functions that currently refer to files (e.g., PRINT, PRIN1, COPYBYTES, FULLNAME) will also accept streams, and will operate slightly more efficiently on them. In addition, the following two functions provide binary input and output on streams:

(BIN *STREAM*) [Function]

Returns the next byte from *STREAM*; thus, this operation is similar to (CHCON1 (READC *STREAM*)). BIN is a very efficient (microcoded) operation.

(BOUT *STREAM BYTE*) [Function]

Outputs a single 8-bit byte to *STREAM*, i.e., similar to (PRIN3 (CHARACTER *BYTE*)).

In addition, the following function coerces files to streams:

(GETSTREAM *FILE ACCESS*) [Function]

Takes a designator which can be used as a "file" argument (e.g., a full/partial file name, a display stream, window, etc.) and returns the corresponding stream. If given a stream will merely return it. *ACCESS* is interpreted the same as in OPENP (page 6.2).

BIN and BOUT will also accept a file designator, in which case they coerce it to a stream via GETSTREAM. However, BIN executes in microcode only when given a stream directly.

18.16.8 Temporary Files and the CORE Device

The local DSK device and most file servers do not support the temporary or scratch files that are available in Interlisp-10. Files that are created do not disappear when some later event such as logout occurs and instead must be deleted by specific action on the part of the user. For this reason, the ;S and ;T suffixes in file names are simply ignored when output is directed to a particular host or device.

However, Interlisp-D does support a notion of core-resident files, and in many cases these provide a reasonable substitute for Interlisp-10 scratch files. Core-resident files are on the device CORE (e.g. {CORE}<FOO>FIE.DCOM;5). The directory for this device and all files on it are represented completely within the user's virtual memory. These files are treated as ordinary files by all file operations; their only distinguishing feature is that all trace of them disappears when the virtual memory is abandoned.

In Interlisp-D, the function PACKFILENAME is defined to default the device name to CORE if the file has the TEMPORARY attribute and no explicit host is provided.

Interlisp-D is initialized with the single core-resident device CORE, but the function COREDEVICE may be used to create any number of logically distinct core devices.

(COREDEVICE *NAME*) [Function]

Creates a new device for core-resident files and assigns *NAME* as its device name. Thus, after performing (COREDEVICE 'FOO), one can execute (OUTFILE '{FOO}BAR) to open a file on that device.

If the directory information associated with CORE devices is not needed, the device NODIRCORE can be used to open core-resident files which "disappear" when they are closed. Note that {NODIRCORE} files do not have names, so the only way to manipulate them is to pass around the value that OPENFILE returned when the file was opened.

18.16.9 Floppy Disks on the Xerox 1108

Interlisp-D on the Xerox 1108 can access the built-in floppy disk drive as device {FLOPPY}. The floppy format is compatible with the Pilot floppy disk format.

18.16.10 Page Mapping

Interlisp-D implements the page-mapping primitives of Interlisp-10 with some notable differences that might require major reworking of programs that rely on these facilities (see page 14.17). The major difference is that an Interlisp-D page contains 256 16-bit words, rather than the 512 36-bit words of Interlisp-10. A given page number or file address for MAPPAGE or MAPWORD will correspond to a very different number of bits from the beginning of the file, and WORDCONTENTS and SETWORDCONTENTS move smaller amounts of information. A second difference is that buffers are completely integrated into the Interlisp-D storage management system so that a page is guaranteed to be locked down as long as the user holds a pointer to it. The functions LOCKMAP and UNLOCKMAP are therefore unnecessary, but for compatibility are defined with dummy definitions.

18.17 FILE SERVERS

A file server is a shared resource on a local communications network which provides large amounts of file storage. Different file servers honor a variety of access protocols. In order to support full Lisp I/O, a file server must provide a random access protocol. One such protocol is Leaf. It has been integrated into the Interlisp-D file system to allow files on a file server to be treated in much the same way files are accessed on the local disk. Except where noted in this section, the standard file operations (OPENFILE, INFILEP, CLOSEF, etc.) all work for remote files. This section explains how to make use of remote files and what differences exist between them and other files.

18.17.1 File Server File Names

The full name of a file on a file server host includes the name of the host in braces, and a directory specification in angle brackets, e.g., {PHYLUM}<LISP>FOO.DCOM;3. These names are not necessarily the syntax by which the actual device/server knows the files (e.g. some file servers use "!" instead of ";"), but Lisp presents a uniform set of naming conventions.

The user can "connect" to a directory on a file server using the CONN command (page 18.11), after which any filename supplied that does not include the host name and/or directory will use the "connected" host and/or directory. Specifically, if the host is omitted, then the connected host is used, and if the directory is also omitted, the connected directory is used as well. If an explicit host is supplied, no defaulting of the directory occurs.

Interlisp supports a preliminary version of NS filing to Xerox 8030 file servers (see page 21.13). Any device with a colon in its name is presumed to be accessible with NS protocols rather than PUP, e.g., {STARFILE:}. The general format of NS fileserver device names is {*SERVER*:*DOMAIN*:*ORGANIZATION*}; the device specification for an 8000-series product must contain the ClearingHouse domain and organization, but if not supplied directly, then they are obtained from the defaults, which themselves are found by a search for the nearest ClearingHouse. NS file servers are modeled after the Star world, and have "File Drawers" rather than directories; "File Folders" are like sub-directories. The functions DIRECTORY, FILEBROWSER, INFILE, COPYFILE, LOAD, and MAKEFILE are working now with NS file servers.

NETWORKOSTYPES [Variable]

Files servers on different machines have different login protocols, file name formats, etc. For proper service from file servers other than Xerox file servers, the user should add entries to the association-list NETWORKOSTYPES associating the host name (all uppercase) with its operating system type, currently one of TENEX, TOPS20, UNIX, or VMS. For example (ADDTOVAR NETWORKOSTYPES (MAXC2 . TENEX)) will inform Interlisp that the file server MAXC2 is a TENEX file server.

18.17.2 Logging In

Most file servers require a user name and password for access. When a file server requests this information, Interlisp-D first gives the name and password from the Alto Executive. If the file server doesn't recognize that name/password, Interlisp-D prompts the user for a name and password to use. It suggests a default name (the one on the disk), which the user can accept by typing a space, or replace by typing a new name or backspacing over it. Interlisp-D saves names and passwords for each host, so the user can login to different file servers using different names.

(LOGIN *HOSTNAME* — — —) [Function]

Forces Interlisp-D to ask for the login name and password to be used when accessing host *HOSTNAME*. Any previous login information for *HOSTNAME* is overriden. If *HOSTNAME* is NIL, it overrides login information for all hosts. Password information vanishes when LOGOUT, SYSOUT, or MAKESYS is executed. Returns the login user name.

18.17.3 Abnormal Conditions

If Interlisp-D tries to access a file and does not get a response from the file server in a reasonable period of time, it prints a message that the file server is not responding, and keeps trying. If the file server has actually crashed, this may continue indefinitely. A CTRL-E or similar interrupt aborts out of this state.

If the file server crashes but is restarted before the user attempts to do anything, file operations will usually proceed normally, except for a brief pause while Interlisp-D tries to reestablish any connections it had open before the crash. It will inform the user of any problems that arise in so doing. The most likely problem occurs when a file has been opened for output but has not yet been written to (or not enough has been written so that Interlisp-D has written to the file server). In this case the file server will think the file is not there when Interlisp-D tries to reestablish the connection. A similar situation arises if the system has been idle (or at least has not accessed the file server) for a sufficiently long period. In this case, the file server will time out the connection. Normally, Interlisp-D will attempt to recover gracefully as described above.

LOGOUT closes any Leaf connections that are currently open. On return, it attempts to reestablish connections for any files that were open before logging out. If a file has disappeared or been modified, Interlisp-D reports this fact.

If it is desired to break the Leaf connection without logging out, call (BREAKCONNECTION *HOST*). Any subsequent reference to files on that host will reestablish the connection. The main reason for doing this occurs if Interlisp-D is interrupted while a file is being opened, leaving the file server thinking the file is open and Lisp thinking it is closed, and then getting a file busy when Interlisp-D next tries to open it.

On rare occasions, the Ethernet may appear completely unresponsive, due to Interlisp having gotten into a bad state. Typing (RESTART.ETHER) will reinitialize Lisp's Ethernet driver(s), just as when the Lisp system is started up following a LOGOUT, SYSOUT, etc (see page 21.15)

18.17.4 Caveats

Leaf does not currently support directory enumeration except for one minor case (in the version field). Hence, DIRECTORY or FILDIR cannot be used on a Leaf file server to get a list of files.

INFILEP and GETFILEINFO currently have to open the file for input in order to obtain their information, and hence the file's read date will change, even though the semantics of these functions do not imply it. This differs from the operation of DSK, and from Interlisp-10 file operations.

Interlisp supports simultaneous access to the same server from different processes and permits overlapping of Lisp computation with file server operations, allowing for improved performance. However, as a corollary of this, a file is not closed the instant that CLOSEF returns; Interlisp closes the file "in the background". It is therefore very important that the user exits Interlisp via (LOGOUT), or (LOGOUT T), rather than boot the machine or exit via Raid.

18.17.5 New Functionality

Certain file servers treat text and binary files differently. Files on file servers can have the attribute TYPE, with value TEXT or BINARY, for use with GETFILEINFO and SETFILEINFO. The file type defaults to the value of DEFAULTFILETYPE, initially TEXT. OPENFILE accepts (TYPE TEXT) or (TYPE BINARY) as an element of its argument *MACHINE.DEPENDENT.PARAMETERS*.

Another allowed element of *MACHINE.DEPENDENT.PARAMETERS* is DON'T.CHANGE.DATE, which means not to change the file's creation date when a file is opened (meaningful only for files being opened for output).

Interlisp-D includes an implementation of the PupFtp protocol, which supports transferring files sequentially only. In those cases where sequential access (as opposed to random access) to a file is appropriate, the use of PupFtp generally results in considerable speed improvement over Leaf, particularly for writing files on a Xerox IFS. The system tries to use PupFtp where possible for SYSOUT and for the destination file of a COPYFILE. One can indicate that a file is going to be accessed only sequentially by including the keyword SEQUENTIAL in the list of *MACHINE.DEPENDENT.PARAMETERS* passed to OPENFILE; the PupFtp will be used, if possible. If for some reason your file server supports PupFtp but you do not wish COPYFILE or SYSOUT to use it, you can set the internal variable \FTPAVAILABLE to NIL.

18.18 HARDCOPY FACILITIES

Note: The following implementation of hardcopy facilities is subject to change.

Interlisp-D includes facilities for generating hardcopy in both "Press" and "Interpress" formats. "Press" is a file format used for communicating documents to laser Xerographic printers called "Dover" (at MIT, Stanford, and CMU) or "Penguin" (everywhere else). "Interpress" is a Xerox standard format used by the 8044 printer and other Network System printers. The hardcopy functions below will generate Press or Interpress output depending on the setting of the function PRINTERMODE:

(PRINTERMODE *x*) [Function]

Sets the type of printing file format generated by LISTFILES, HARDCOPYW, and printer devices (see PRINTERDEVICE, below). If *x* is PRESS, the Press file format is used. If *x* is INTERPRESS, the Interpress file format is used.

Currently, the hardcopy interface is not smart enough to infer the printer mode from a previously formatted file or the name of a printing host. If the user wants to print a previously formatted Press or Interpress file, the printing mode must be set correctly.

(PRINTINGHOST —) [Function]

The function PRINTINGHOST is used to find the name of the local printer.

For (PRINTERMODE 'PRESS), this merely returns the value of the variable DEFAULTPRINTINGHOST, which is usually set by an entry in the site greeting file (see page 14.5).

For `(PRINTERMODE 'INTERPRESS)`, this returns the value of the variable `NS.DEFAULT.PRINTER` if non-`NIL`, otherwise it returns the first local printer found in the closest clearinghouse (see page 21.11).

The function `LISTFILES1` is used by `LISTFILES` to send a single file to a hardcopy printing device. Interlisp-D is initialized with `LISTFILES1` defined to call `EMPRESS` in Press mode or `NSPRINT` (page 21.11) in Interpress mode. These functions convert a file to Press or Interpress format and send it to a printing server. The "default" site greeting file delivered with the Xerox 1100 redefines `LISTFILES1` as a no-op.

`(EMPRESS` *FILE #COPIES HOST HEADING #SIDES*`)` [Function]

The function `EMPRESS` causes *#COPIES* copies of the file *FILE* to be sent to the printer *HOST*. If *HOST* is `NIL`, the value of `(PRINTINGHOST)` is used. *#SIDES* specifies one- or two-sided printing; may be 1 or 2 (if *HOST* is capable of duplex printing) or `T` (meaning to use the printer's default); defaults to the value of `EMPRESS#SIDES`, initially `T`.

If *FILE* is a Press or Interpress format file, it is transmitted directly. Otherwise, it is converted by calling the function `MAKEPRESS` (called with *FONTS* = `NIL` and the same *HEADING*).

`EMPRESS.SCRATCH` [Variable]

`EMPRESS` constructs scratch press files on the `{CORE}` device for small files. If the number of disk pages of the source file is larger than the limit set by the first element of the list `EMPRESS.SCRATCH`, an alternate scratch file, specified by the second element of `EMPRESS.SCRATCH`, is used. `EMPRESS.SCRATCH` is initialized to `(30 {DSK}EMPRESS.SCRATCH)`.

`(MAKEPRESS` *FILE OUTFILE FONTS HEADING TABS*`)` [Function]

`(MAKEINTERPRESS` *FILE OUTFILE FONTS HEADING TABS*`)` [Function]

These functions produce a Press or Interpress file named *OUTFILE* from the `ASCII` file *FILE*. If *OUTFILE* is `NIL`, it defaults to the same file name as *FILE*, with extension `Press` or `Interpress`.

These functions interpret character sequences beginning with control-F (character code 6) as special formatting instructions. If the code of the next character is a valid font number, then the formatting sequence indicates a change to that font. The correspondence between font numbers and fonts is specified by entries on the list *FONTS* or, if *FONTS* is `NIL`, the current font profile list (see page 6.55). Each entry is of the form `(`*FONT/CLASS FONT-NUMBER DISPLAY-FONT PRESS-FONT*`)`. For example, the entry `(DEFAULTFONT 1 (GACHA 10) (GACHA 8))` indicates that `GACHA 8` will be used in press files for font 1 which will be represented on the display as `GACHA 10`. *HEADING* is a string that is printed as a heading on each page. If *HEADING* is `NIL`, the file's name and creation date will be used.

These functions also allow absolute tab stops to be specified. If the control-F is followed by a control-T, the code of the character after that is interpreted as an absolute tab stop number. The corresponding entry on the list *TABS*, or `PRESSTABSTOPS` if *TABS* is `NIL`, is taken as the number of mills from the left margin at which printing on the current line will continue. `PRESSTABSTOPS` is initially `(8000)`.

FONTWIDTHSFILES [Variable]

Value is a file name or a list of file names to be searched for information about the widths of characters in particular fonts. This variable should be initialized in the site greeting file.

(HARDCOPYW *WINDOW/BITMAP/REGION FILE HOST SCALEFACTOR ROTATION*) [Function]

Creates bitmap hardcopy and optionally sends it to a printer. *WINDOW/BITMAP/REGION* can either be a WINDOW (open or closed), a BITMAP, or a REGION (interpreted as a region of the screen). If NIL, the user is prompted for a region using GETREGION (page 19.37) in a manner which "defaults" to the whole screen.

The logic of defaulting is complex and follows:

FILE, if supplied, will be used as the name of the file for output. If *HOST* is NIL, then if *FILE* was given, no printing is performed, else if FULLPRESSPRINTER is non-NIL, then output is sent to that printer, else output is sent to the value of (PRINTINGHOST). To save an image on a file without printing it, perform (HARDCOPYW *IMAGE FILE*).

SCALEFACTOR is a reduction factor. Only *SCALEFACTOR*=1 can be printed on Dover and Penquin printers. *SCALEFACTOR* defaults according to the size of the image, the size of a page, and the parameters *HOST*, *FILE*, and FULLPRESSPRINTER in a complex but appropriate manner.

ROTATION, which can be one of 0, 90, 180, 270 (default 0) specifies how the bitmap image should be rotated on the printed page. This may not be supported by some printers.

Note that "Hardcopy" in the background menu merely performs (HARDCOPYW), which sends an image of region user selects to the default printer. Hardcopy in the paint menu performs (HARDCOPYW *WINDOW*), which sends an image of window to the default printer.

(PRESSFILEP *FILE*) [Function]

Returns (FULLNAME *FILE*) if *FILE* is a Press file, NIL otherwise.

Hardcopy output may also be obtained by writing a file on the printer device LPT, e.g. (COPYFILE 'FOO '{LPT}). When a file on this device is closed, it is converted to Press or Interpress format (if necessary) and sent to the default printer. Thus, {LPT} acts like the device LPT: in Interlisp-10. Printer devices can be defined for other network printer hosts with the following function:

(PRINTERDEVICE *NAME*) [Function]

Defines the network printer host *NAME* to be a printer device treated like LPT. For example, if (PRINTERDEVICE 'YODA) is executed, then (COPYFILE 'FOO '{YODA}) will transmit FOO to the printer named YODA.

18.19 PERFORMANCE CONSIDERATIONS

Most Interlisp-D users will have experience using Interlisp-10. Although Interlisp-D is completely upward compatible with Interlisp-10, there are differences in the exact implementation which may influence the performance of applications programs. This chapter contains a collection of notes which may help the user improve the performance of Interlisp-D programs.

18.19.1 Variable Bindings

A major difference between Interlisp-10 and Interlisp-D is the method of accessing free variables. Interlisp-10 uses what is called "shallow" binding. Interlisp-D uses what is called "deep" binding.

The binding of variables occurs when a function or a PROG is entered. For example, if the function FOO has the definition (LAMBDA (A B) *BODY*), the variables A and B are bound so that any reference to A or B from *BODY* or any function called from *BODY* will refer to the arguments to the function FOO and not to the value of A or B from a higher level function. All variable names (atoms) have a top level value cell which is used if the variable has not been bound in any function. In discussions of variable access, it is useful to distinquish between three types of variable access: local, special and global. Local variable access is the use of a variable that is bound within the function from which it is used. Special variable access is the use of a variable that is bound by another function. Global variable access is the use of a variable that has not been bound in any function. We will often refer to a variable all of whose accesses are local as a "local variable." Similarly, a variable all of whose accesses are global we call a "global variable."

In a "deep" bound system, a variable is bound by saving on the stack the variable's name together with a value cell which contains that variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding (occurrence) and retrieving the value stored there. If the variable is not found on the stack, the variable's top level value cell is used.

In a "shallow" bound system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's top level value cell. When a variable is accessed, its value is always found in its top level value cell.

The deep binding scheme has one disadvantage: the amount of cpu time required to fetch the value of a variable depends on the stack distance between its use and its binding. The compiler can determine local variable accesses and compiles them as fetches directly from the stack. Thus this computation cost only arises in the use of variable not bound in the local frame ("free" variables). The process of finding the value of a free variable is called free variable lookup.

In a shallow bound system, the amount of cpu time required to fetch the value of a variable is constant regardless of whether the variable is local, special or global. The disadvantages of this scheme are that the actual binding of a variable takes longer (thus slowing down function call), the cells that contain the current in use values are spread throughout the space of all atom value cells (thus increasing the working set size of functions) and context switching between processes requires unwinding and rewinding the stack (thus effectively prohibiting the use of context switching for many applications).

A deep binding scheme was choosen for Interlisp-D because of the working set considerations and the speed of context switching, which we expected to use heavily when processes were added. The free variable lookup routine was microcoded, thus greatly reducing the search time. In the benchmarks we performed, the largest percentage of free variable lookup time was 20 percent of the total ellapsed time; the normal time was between 5 and 10 percent.

One consequence of Interlisp-D's deep binding scheme is that users may significantly improve performance by declaring global variables in certain situations. If a variable is declared global, the compiler will compile an access to that variable as a retrieval of its top level value, completely bypassing a stack search. This should be done only for variables that are never bound in functions such as global databases and flags.

Global variable declarations should be done using the GLOBALVARS file package command (page 11.25). Its form is (GLOBALVARS VAR_1 ··· VAR_N).

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form (DECLARE (LOCALVARS VAR_1 ··· VAR_N)) following the argument list in the definition of the function. Note: local variable declarations only effect the compilation of a function. Interpreted functions put all of their variable names on the stack, regardless of any declarations.

18.19.2 Garbage Collection

As an Interlisp-D applications program runs, it creates data structures (allocated out of free storage space), manipulates them, and then discards them. If there were no way of reclaiming this space, over time the Interlisp-D memory (both the physical memory in the machine and the virtual memory stored on the disk) would get filled up, and the computation would come to a halt. Actually, long before this would happen the system would probably become intolerably slow, due to "data fragmentation", which occurs when the data currently in use are spread over many virtual memory pages, so that most of the computer time must be spent swapping disk pages into physical memory. This problem ("fragmentation") will occur in any situation where the virtual memory is significantly larger than the real, physical memory. To reduce swapping, it is desirable to keep the "working set" (the set of pages containing actively referenced data) as small as possible.

It is possible to write programs that don't generate much "garbage" data, or which recycle data, but such programs tend to be overly complicated and frought with pitfalls. Spending effort writing such programs defeats the whole point of using a system with automatic storage allocation. An important part of any Lisp implementation is the "garbage collector" which identifies discarded data and reclaims its space. There are several well-known approaches to garbage collection. Interlisp-10 uses the traditional mark-and-sweep garbage collection algorithm, which identifies "garbage" data by "walking" through and "marking" all accessible data structures, and then sweeping through the data spaces to find all unmarked objects (i.e., not referenced by any other object). Although this method is guaranteed to reclaim all garbage, it takes time proportional to the number of allocated objects, which may be very large. (Some allocated objects will have been marked during the "mark" phase, and the remainder will be collected during the "sweep" phase; so all will have to be touched in some way.) Also, the time that a mark-and-sweep garbage collection takes is independent of the amount of garbage collected; it is possible to sweep through the whole virtual memory, and only recover a small amount of garbage.

For interactive applications, it is simply not acceptable to have long interruptions in a computation for the purpose of garbage collection. Interlisp-D solves this problem by using a reference-counting garbage collector. With this scheme, there is a table containing counts of how many times each object is referenced. This table is incrementally updated as pointers are created and discarded, incurring a small overhead distributed over the computation as a whole. (Note: References from the stack are not counted, but are handled separately at "sweep" time; thus the vast majority of data manipulations do not cause updates to this table.) At opportune moments, the garbage collector scans this table, and reclaims all objects that are no longer accessible (have a reference count of zero). The time for scanning the reference count tables is very nearly constant (about 0.2 seconds on the Xerox 1100); the sweep time then is this small value, plus time proportional to the amount of garbage that has to be collected (typically less than a second). "Opportune" times occur when a certain number of cells have been allocated or when the system has been waiting for the user to type something for long enough. The frequency of garbage collection is controlled by the functions and variables described on page 18.2. For the best system performance, it is desirable to adjust these parameters for frequent, short garbage collections, which will not interrupt interactive applications for very long, and which will have the added benefit of reducing data fragmentation, keeping the working set small.

One problem with the Interlisp-D garbage collector is that not all garbage is guaranteed to be collected. Circular data structures, which point to themselves directly or indirectly, are never reclaimed, since their reference counts are always at least one. With time, this unreclaimable garbage may increase the working set to unacceptable levels. Some users have worked with the same Interlisp-D virtual memory for a very long time, but it is a good idea to occasionally save all of your functions in files, reinitialize Interlisp-D, and rebuild your system. Many users end their working day by issuing a command to rebuild their system and then leaving the machine to perform this task in their absence. If the system seems to be spending too much time swapping (an indication of fragmented working set), this procedure is definitely recommended.

18.19.3 Datatypes

If an applications program uses data structures that are large (more than 8 fields) and that are used a lot, there are several advantages to representing them as user DATATYPEs rather than as RECORDs. The primary advantage is increased speed: accessing and setting the fields of a DATATYPE can be significantly faster than walking through a RECORD list with repeated CARs and CDRs. Also, compiled code for referencing user DATATYPEs is usually smaller. Finally, by reducing the number of objects created (one DATATYPE object against many RECORD list cells), this can reduce the expense of garbage collection.

For code that has been written using the record package's fetch, replace, and create operations, changing from RECORDs to DATATYPEs only requires editing the record declaration (using EDITREC) to replace declaration type RECORD by DATATYPE, and recompiling.

18.19.4 Incomplete Filenames

There is a significant problem in Interlisp-D (and in Interlisp-10) with respect to using incomplete filenames. Whenever an I/O function is given an incomplete filename (one which doesn't have the device/host, directory, name, extension, and version number all supplied), the system has to convert it to a complete filename, by supplying defaults and searching through directories (which may be on remote file servers). Currently, work is being done on speeding up the filename-completion process, but in any case it is much faster to convert an incomplete filename once, and use the complete filename from then on. For example, suppose a file is opened with (SETQ FULLNAME (OPENFILE 'MYNAME 'INPUT)). After doing this, (READC 'MYNAME) and (READC FULLNAME) would both work, but (READC 'MYNAME) would take longer (sometimes orders of magnitude longer). This could seriously effect the performance if a program which is doing many I/O operations.

18.19.5 Turning Off the Display

Maintaining the video image on the screen uses about 30% of the cpu cycles (on the Xerox 1100), so turning off the display will improve the speed of compute-bound tasks. When the display is off, the screen will be white but any printing or displaying that the program does will be visible when the display is turned back on. Note: Breaks and PAGEFULLFN waiting turn the display on, but users should be aware that it is possible to have the system waiting for a response to a question printed or a menu displayed on a non-visible part of the screen. The following functions are provided to turn the display off:

(SETDISPLAYHEIGHT *NSCANLINES*) [Function]

Sets the display to only show the top *NSCANLINES* of the screen. If *NSCANLINES* is T, resets the display to show the full screen. Returns the previous setting.

(DISPLAYDOWN *FORM NSCANLINES*) [Function]

Evaluates *FORM* (with the display set to only show the top *NSCANLINES* of the screen), and returns the value of *FORM*. It restores the screen to its previous setting. If *NSCANLINES* is not given, it defaults to 0.

18.19.6 Gathering Statistics

Interlisp-D has an extended set of statistics-gathering tools. An extended version of the TIME function is provided:

(TIMEALL *TIMEFORM #TIMES TIMEWHAT INTERPFLG* —) [NLambda Function]

Largely subsumes the function TIME. Evaluates the form *TIMEFORM* and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and datatype allocation.

For more accurate measurement on small computations, *#TIMES* may be specified (its default is 1) to cause *TIMEFORM* to be executed *#TIMES* number of times. To improve the accuracy of timing open-coded operations in this case, TIMEALL compiles a form to execute *TIMEFORM #TIMES* number of times (unless *INTERPFLG* is non-NIL), and then times the execution of the compiled form. The compilation is with optimizations off to avoid constant folding.

TIMEWHAT exists largely for compatibility with TIME; it restricts the statistics to specific categories. It can be an atom or list of datatypes to monitor, and/or the atom TIME to monitor time spent. Note that ordinarily, TIMEALL monitors all time and datatype usage, so this argument is rarely needed.

The value of TIMEALL is the value of the last evaluation of *TIMEFORM*.

The Interlisp-D system has a facility for gathering very low-level statistics on function call and return. It is conceptually like performing a BREAKDOWN on every function in the world. The system designers regularly use this facility to determine where time is being spent in suspect computations, suggesting which parts of the system code deserve optimizing.

(DOSTATS *FORM TITLE* — — —) [Function]

Collects statistics of the evaluation of *FORM* and produces a listing of the results. *TITLE*, if supplied, will appear in the heading of the listing.

Performing a statistics run consists of three phases:

Gathering

The microcode is instructed to emit a statistics event for every function call and return that is executed, and *FORM* is evaluated. These events are collected on a file for the next phase (the name of the file is {DSK}*xxx*.STATS, where *xxx* = (CAR *FORM*)). Currently the file must reside on {DSK}, so be sure you have a lot of space. Even seemingly short computations can generate large numbers of function call/return events. If your disk fills up, Lisp may not recover gracefully (it usually falls into SWAT).

Analysis

The statistics file is read in. For each event, a counter associated with the indicated function is incremented by the amount of time spent in the function. The analysis also records who called which functions, how often, and with how many arguments. This is by far the longest phase.

Summarizing

The results of the analysis are used to produce a listing that shows each of the functions called, sorted by their contribution to the total time, and a cross-reference of who called whom. The listing is put on a file *xxx*.PRINTOUT on the connected directory and also shipped to your local printer.

Excerpts from a sample statistics printout are shown below, with commentary. The form is (RECLAIM), which was fairly brief in this case.

Notes

The times shown in the printout are for time spent in a single function; there is no cumulative time measurement. The percentages should thus add up to 100%. If FOO calls FIE, the time spent inside FIE is charged to FIE only, not to FOO as well.

The times recorded are of the right order of magnitude, and can be compared to each other, but should not be taken literally, as they are inflated by the overhead of recording each call and return event. The total elapsed time for the evaluation phase is much larger still, being dominated by the time to dump the statistics to disk, but this part of the time is filtered out in the analysis.

```
Statistics from file: {DSK}RECLAIM.STATS;1

measuring: evaluation of
FORM = (RECLAIM)

Computation run on Dolphin serial #237 with 2304 pages of memory.
Versions:  Ram=7401(17,1)  Bcpl=17400(37,0)  Lisp=106000(214,0)
           (Internal version numbers of microcode, Lisp.run, Lisp.sysout) .

Unrecognized events: NIL (everything was okay)

Values from MiscStats (times in msecs):
      SWAPWAITTIME           6137
      PAGEFAULTS               58
      GCTIME                27392

Not Windowing
```

```
Filtering out \StackOverflow, \NWWInterrupt, \PageFault, \StatsOverflow
```

(time for these functions measured separately)

```
Ignoring time for GETKEYS, \GETKEY, WAITFORINPUT, DISMISS, GATHERSTATS,
                  \GATHERSTATS, RAID
```

(time for these functions ignored completely)

```
Function timings:                         #ofCalls   PerCall
 total time spent in function (microseconds)
 |         percentage of total analyzed time spent in function
 |         |          function name. Number of arguments in brackets
 |         |          |                    number of calls recorded to this fn
 |         |          |                    |     avg time per call (microseconds)
 1746426  36.08%  \GCMAPTABLE [1]          524      3332
 1104420  22.81%  \GCMAPSCAN [0]             1   1104420
  794862  16.42%  \HTFIND [2]             1236       643
  461194   9.52%  \FREELISTCELL [1]       2044       225
  457537   9.45%  \GCRECLAIMCELL [1]      1533       298
   77437   1.59%  \GCMAPUNSCAN [0]           1     77437
   52907   1.09%  RELEASINGVMEMPAGE [1]     30      1763
   47308   0.97%  \GCSCANSTACK [0]           1     47308
   45365   0.93%  FINDPTRSBUFFER [2]        30      1512
    9218   0.19%  \ADDBASE [2]              31       297
    7618   0.15%  CREATECELL [1]            18       423
    7428   0.15%  \INSERTBLOCK [1]          31       239
    6856   0.14%  \RECLAIMARRAYBLOCK [1]    31       221
   21597   0.44%  for 18 entries not shown
                   (functions contributing less than .1% are omitted)
 4840173          Total for 31 entries    5511
```

```
Function timings: Filtered out fns   #ofCalls   PerCall
```

(times for functions whose contribution was omitted from the analysis above)

```
20225828  70.32%  Subr.\StatsOverflow [0] 413     48972  (stats overhead)
 6900042  23.99%  Subr.\PageFault [1]       58    118966  (pagefault activity)
 1635737   5.68%  Subr.\NWWInterrupt [0]  762      2146  (periodic service)
28761607          Total for 3 entries    1291
```

```
Function timings: Alphabetic         #ofCalls   PerCall
```

(listing as above, but including all functions, and sorted alphabetically)

. . .

```
Call Information:
```

(Alphabetic listing of functions, with calls and callers information)

(number of calls in parentheses)

```
CLOCK
          Calls:   MAKENUMBER (8), \SLOWIPLUS2 (6), CLOCK (2),
                   CREATECELL (2), CLOCK0 (2), \SLOWIDIFFERENCE (2)
          Callers: \DORECLAIM (2), CLOCK (2)
CLOCK0
          Callers: CLOCK (2)
CREATECELL
          Calls:   \HTFIND (1)
          Callers: MAKENUMBER (16), CLOCK (2)

       . . .
```

18.20 THE INTERLISP-D PROCESS MECHANISM

The Interlisp-D Process mechanism provides an environment in which multiple Lisp processes can run in parallel. Each executes in its own stack space, but all share a global adress space. The current process implementation is cooperative; i.e., process switches happen voluntarily, either when the process in control has nothing to do or when it is in a convenient place to pause. There is no preemption or guaranteed service, so you cannot run something demanding (e.g., Chat) at the same time as something that runs for long periods without yielding control. Keyboard input and network operations block with great frequency, so processes currently work best for highly interactive tasks (editing, making remote files).

In Interlisp-D, the process mechanism is already turned on, and is expected to stay on during normal operations, as some system facilities (in particular, most network operations) require it. However, under exceptional conditions, the following function can be used to turn the world off and on:

(PROCESSWORLD *FLG*) [Function]

Starts up the process world, or if *FLG* = OFF, kills all processes and turns it off. Normally does not return. The environment starts out with two processes: a top-level EVALQT (the initial "tty" process) and the "background" process, which runs the window mouse handler and other system background tasks.

Note: PROCESSWORLD is intended to be called at the top level of Interlisp, not from within a program. It does not toggle some sort of switch; rather, it constructs some new processes in a new part of the stack, leaving any callers of PROCESSWORLD in a now inaccessible part of the stack. Calling (PROCESSWORLD 'OFF) is the only way the call to PROCESSWORLD ever returns.

(HARDRESET) [Function]

Resets the whole world, and rebuilds the stack from scratch. This is "harder" than doing RESET to every process, because it also resets system internal processes (such as the keyboard handler).

HARDRESET automatically turns the process world on (or resets it if it was on), unless the variable AUTOPROCESSFLG is NIL.

18.20.1 Creating and Destroying Processes

(ADD.PROCESS *FORM PROP$_1$ VALUE$_1$ ··· PROP$_N$ VALUE$_N$*) [NoSpread Function]

Creates a new process evaluating *FORM*, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which ADD.PROCESS was called; all such information must be passed as arguments in *FORM*. The process runs until *FORM* returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its RESTARTABLE property is T), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to PROCESSPROP may be given, but the following two are directly relevant to ADD.PROCESS:

NAME Value should be a litatom; if not given, the process name is taken from (CAR *FORM*). ADD.PROCESS may pack the name with a number to make it unique. This name is solely for the convenience of manipulating processes at Lisp typein; e.g., the name can be given as the *PROC* argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.

SUSPEND

If the value is non-NIL, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a WAKE.PROCESS (below).

(PROCESSPROP *PROC PROP NEWVALUE*) [NoSpread Function]

Used to get or set the values of certain properties of process *PROC*, in a manner analogous to WINDOWPROP. If *NEWVALUE* is supplied (including if it is NIL), property *PROP* is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

NAME Value is a litatom used for identifying the process to the user.

RESTARTABLE

Value is a flag indicating the disposition of the process following errors or hard resets:

NIL or NO

(the default) If an untrapped error (or control-E or control-D) causes its form to be exited, the process is deleted. The process is also deleted if a HARDRESET (or control-D from RAID) occurs, causing the entire Process world to be reinitialized.

T or YES

The process is automatically restarted on errors or HARDRESET. This is the normal setting for persistent "background" processes, such as the mouse process, that can safely restart themselves on errors.

HARDRESET

The process is deleted as usual if an error causes its form to be exited, but it *is* restarted on a HARDRESET. This setting is preferred for persistent processes for which an error is an unusual condition, one that might repeat itself if the process were simply blindly restarted.

FORM Value is the Lisp form used to start the process (readonly).

AFTEREXIT

Value indicates the disposition of the process following a resumption of Lisp after some exit (LOGOUT, SYSOUT, MAKESYS). Possible values are:

DELETE

Delete the process.

SUSPEND

Suspend the process; i.e., do not let it run until it is explicitly woken.

<an event>

Cause the process to be suspended waiting for the event (page 18.30).

INFOHOOK

Value is a function or form used to provide information about the process, in conjunction with the process status window (page 18.36).

WINDOW

Value is a window associated with the process, the process's "main" window. Used in conjunction with switching the tty process (page 18.33).

TTYENTRYFN

Value is a function that is applied to the process when the process is made the tty process (page 18.33).

TTYEXITFN

Value is a function that is applied to the process when the process ceases to be the tty process (page 18.33).

(THIS.PROCESS) [Function]

Returns the handle of the currently running process, or NIL if the Process world is turned off.

(DEL.PROCESS *PROC* —) [Function]

Deletes process *PROC*. *PROC* may be a process handle (returned by ADD.PROCESS), or its name. Note that if *PROC* is the currently running process, DEL.PROCESS does not return!

(PROCESS.RETURN *VALUE*) [Function]

Terminates the currently running process, causing it to "return" *VALUE*. There is an implicit PROCESS.RETURN around the *FORM* argument given to ADD.PROCESS, so that normally a process can finish by simply returning; PROCESS.RETURN is supplied for earlier termination.

(PROCESS.RESULT *PROCESS* *WAITFORRESULT*) [Function]

If *PROCESS* has terminated, returns the value, if any, that it returned. This is either the value of a PROCESS.RETURN or the value returned from the form given to ADD.PROCESS. If the process was aborted, the value is NIL. If *WAITFORRESULT* is true, PROCESS.RESULT blocks until *PROCESS* finishes, if necessary; otherwise, it returns NIL immediately if *PROCESS* is still running. Note that *PROCESS* must be the actual process handle returned from ADD.PROCESS, not a process name, as the association between handle and name disappears when the process finishes (and the process handle itself is then garbage collected if no one else has a pointer to it).

(PROCESS.FINISHEDP *PROCESS*) [Function]

True if *PROCESS* has terminated. The value returned is an indication of how it finished: NORMAL or ERROR.

(PROCESSP *PROC*) [Function]

True if *PROC* is the handle of an active process, i.e., one that has not yet finished.

(RELPROCESSP *PROCHANDLE*) [Function]

True if *PROCHANDLE* is the handle of a deleted process. This is analogous to RELSTKP. It differs from PROCESS.FINISHEDP in that it never causes an error, while PROCESS.FINISHEDP can cause an error if its *PROC* argument is not a process at all.

(RESTART.PROCESS *PROC*) [Function]

Unwinds *PROC* to its top level and reevaluates its form. This is effectively a DEL.PROCESS followed by the original ADD.PROCESS.

(MAP.PROCESSES *MAPFN*) [Function]

Maps over all processes, calling *MAPFN* with three arguments: the process handle, its name, and its form.

(FIND.PROCESS *PROC* *ERRORFLG*) [Function]

If *PROC* is a process handle or the name of a process, returns the process handle for it, else NIL. If *ERRORFLG* is T, generates an error if *PROC* is not, and does not name, a live process.

18.20.2 Process Control Constructs

(BLOCK *MSECSWAIT* *TIMER*) [Function]

Yields control to the next waiting process, assuming any is ready to run. If *MSECSWAIT* is specified, it is a number of milliseconds to wait before returning (in which case BLOCK is very much like DISMISS), or T, meaning wait forever (until explicitly woken). Alternatively, *TIMER* can be given as a millisecond timer (as returned by SETUPTIMER) of an absolute time at which to wake up. In any of those cases, the process enters the *waiting* state until the time limit is up. BLOCK with no arguments leaves the process in the *runnable* state, i.e., it returns as soon as every other runnable process of the same priority has had a chance.

(WAKE.PROCESS *PROC STATUS*) [Function]

Explicitly wakes process *PROC*, i.e., makes it *runnable*, and causes its call to BLOCK (or other waiting function) to return *STATUS*. This is one simple way to notify a process of some happening; however, note that if WAKE.PROCESS is applied to a process more than once before the process actually gets its turn to run, it sees only the latest *STATUS*.

(SUSPEND.PROCESS *PROC*) [Function]

Blocks process *PROC* indefinitely, i.e., *PROC* will not run until it is woken by a WAKE.PROCESS.

The following three functions allow access to the stack context of some other process. They require a little bit of care, and are computationally non-trivial, but they do provide a more powerful way of manipulating another process than WAKE.PROCESS allows.

(PROCESS.EVALV *PROC VAR*) [Function]

Performs (EVALV *VAR*) in the stack context of *PROC*.

(PROCESS.EVAL *PROC FORM WAITFORRESULT*) [Function]

Evaluates *FORM* in the stack context of *PROC*. If *WAITFORRESULT* is true, blocks until the evaluation returns a result, else allows the current process to run in parallel with the evaluation. Any errors that occur will be in the context of *PROC*, so be careful. In particular, note that

```
(PROCESS.EVAL PROC '(NLSETQ (FOO)))
```

and

```
(NLSETQ (PROCESS.EVAL PROC '(FOO)))
```

behave quite differently if FOO causes an error. And it is quite permissible to intentionally cause an error in proc by performing

```
(PROCESS.EVAL PROC '(ERROR!))
```

If errors are possible and *WAITFORRESULT* is true, the caller should almost certainly make sure that *FORM* traps the errors; otherwise the caller could end up waiting forever if *FORM* unwinds back into the pre-existing stack context of *PROC*.

(PROCESS.APPLY *PROC FN ARGS WAITFORRESULT*) [Function]

Performs (APPLY *FN ARGS*) in the stack context of *PROC*. Note same warnings as with PROCESS.EVAL.

18.20.3 Events

An "event" is a synchronizing primitive used to coordinate related processes, typically producers and consumers. Consumer processes can "wait" on events, and producers "notify" events.

(CREATE.EVENT *NAME*) [Function]

Returns an instance of the EVENT datatype, to be used as the event argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(AWAIT.EVENT *EVENT TIMEOUT TIMERP*) [Function]

Suspends the current process until *EVENT* is notified, or until a timeout occurs. If *TIMEOUT* is NIL, there is no timeout. Otherwise, timeout is either a number of milliseconds to wait, or, if *TIMERP* is T, a millisecond timer set to expire at the desired time using SETUPTIMER (see page 14.11).

(NOTIFY.EVENT *EVENT ONCEONLY*) [Function]

If there are processes waiting for *EVENT* to occur, causes those processes to be placed in the running state, with *EVENT* returned as the value from AWAIT.EVENT. If *ONCEONLY* is true, only runs the first process waiting for the event (this should only be done if the programmer knows that there can only be one process capable of responding to the event at once).

The meaning of an event is up to the programmer. In general, however, the notification of an event is merely a hint that something of interest to the waiting process has happened; the process should still verify that the conceptual event actually occurred. That is, *the process should be written so that it operates correctly even if woken up before the timeout and in the absence of the notified event.* In particular, the completion of PROCESS.EVAL and related operations in effect wakes up the process in which they were performed, since there is no secure way of knowing whether the event of interest occurred while the process was busy performing the PROCESS.EVAL.

There is currently one class of system-defined events, used with the network code. Each Pup and NS socket has associated with it an event that is notified when a packet arrives on the socket; the event can be obtained by calling (PUPSOCKETEVENT *PUPSOCKET*) or (NSOCKETEVENT *NSOCKET*), respectively.

18.20.4 Monitors

It is often the case that cooperating processes perform operations on shared structures, and some mechanism is needed to prevent more than one process from altering the structure at the same time. Some languages have a construct called a monitor, a collection of functions that access a common structure with mutual exclusion provided and enforced by the compiler via the use of monitor locks. Interlisp-D has taken this implementation notion as the basis for a mutual exclusion capability suitable for a dynamically-scoped environment.

A monitorlock is an object created by the user and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(CREATE.MONITORLOCK *NAME* —) [Function]

Returns an instance of the MONITORLOCK datatype, to be used as the lock argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(WITH.MONITOR *LOCK* . *FORMS*) [Macro]

Evaluates (PROGN . *FORMS*) while owning *LOCK*. Value is the last of *FORMS*. This construct is implemented so that the lock is released even if the form is exited via error (currently implemented with RESETLST). Ownership of a lock is dynamically scoped: if the current process already owns the lock (e.g., if the caller was itself inside a WITH.MONITOR for this lock), WITH.MONITOR is a noop.

(WITH.FAST.MONITOR *LOCK* . *FORMS*) [Macro]

Like WITH.MONITOR, but implemented without the RESETLST. User interrupts (e.g., control-E) are inhibited during the evaluation of *FORMS*.

Programming restriction: the evaluation of *FORMS* must not error (the lock would not be released). This construct is mainly useful when *FORMS* is a small, safe computation that never errors and need never be interrupted.

(MONITOR.AWAIT.EVENT *RELEASELOCK EVENT TIMEOUT TIMERP*) [Function]

For use in blocking inside a monitor. Performs (AWAIT.EVENT *EVENT TIMEOUT TIMERP*), but releases *RELEASELOCK* first, and reobtains the lock (possibly waiting) on wakeup.

Typical use for MONITOR.AWAIT.EVENT: A function wants to perform some operation on *Foo*, but only if it is in a certain state. It has to obtain the lock on the structure to make sure that the state of the structure does not change between the time it tests the state and performs the operation. If the state turns out to be bad, it then waits for some other process to make the state good, meanwhile releasing the lock so that the other process can alter the structure.

```
(WITH.MONITOR FooLock
   (until condition-of-Foo
    do (MONITOR.AWAIT.EVENT FooLock EventFooChanged timeout))
      operate-on-Foo)
```

It is sometimes convenient for a process to have WITH.MONITOR at its top level and then do all its interesting waiting using MONITOR.AWAIT.EVENT. Not only is this often cleaner, but in the present implementation in cases where the lock is frequently accessed, it saves the RESETLST overhead of WITH.MONITOR.

Programming restriction: there must not be an ERRORSET between the enclosing WITH.MONITOR and the call to MONITOR.AWAIT.EVENT such that the ERRORSET would catch an ERROR! and continue inside the monitor, for the lock would not have been reobtained. (The reason for this restriction is that, although MONITOR.AWAIT.EVENT won't itself error, the user could have caused an error with an interrupt, or a PROCESS.EVAL in the context of the waiting process that produced an error.)

On rare occasions it may be useful to manipulate monitor locks directly. The following two functions are used in the implementation of WITH.MONITOR:

(OBTAIN.MONITORLOCK *LOCK DONTWAIT UNWINDSAVE*) [Function]

Takes possession of *LOCK*, waiting if necessary until it is free, unless *DONTWAIT* is true, in which case it returns NIL immediately. If *UNWINDSAVE* is true, performs a RESETSAVE to be unwound when the enclosing RESETLST exits. Returns *LOCK* if *LOCK* was successfully obtained, T if the current process already owned *LOCK*.

(RELEASE.MONITORLOCK *LOCK*) [Function]

Releases *LOCK* if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

When a process is deleted, any locks it owns are released.

18.20.5 Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to BLOCK, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. "State" variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Interlisp-D to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables—the keyboard and the mouse—are particularly idosyncratic, and are discussed in the next section.

The following resources, which are global in Interlisp-10, are allocated per process in Interlisp-D: primary input and output (the streams affected by INPUT and OUTPUT), terminal input and output (the streams designated by the name T), the primary read table and primary terminal table, and dribble files. Thus, each process can print to its own primary output, print to the terminal, read from a different primary input, all without interfering with another process's reading and printing.

Each process begins life with its primary and terminal input/output streams set to a dummy stream. If the process attempts input or output using any of those dummy streams, e.g., by calling (READ T), or (PRINT & T), a tty window is automatically created for the process, and that window becomes the primary input/output and terminal input/output for the process. The default tty window is created at or near the region specified in the variable DEFAULTTTYREGION.

A process can, of course, call TTYDISPLAYSTREAM explicitly to give itself a tty window of its own choosing, in which case the automatic mechanism never comes into play. Calling TTYDISPLAYSTREAM when a process has no tty window not only sets the terminal streams, but also sets the primary input and output streams to be that window, assuming they were still set to the dummy streams.

(HASTTYWINDOWP *PROC*) [Function]

Returns T if the process *PROC* has a tty window; NIL otherwise. If *PROC* is NIL, it defaults to the current process.

Other system resources that are typically changed by RESETFORM, RESETLST, RESETVARS are all global entities. In the multiprocessing environment, these constructs are suspect, as there is no provision for "undoing" them when a process switch occurs. For example, in the current release of Interlisp-D, it is not possible to set the print radix to 8 inside only one process, as the print radix is a global entity.

Note that RESETFORM and similar expressions are perfectly valid in the process world, and even quite useful, when they manipulate things strictly within one process. The process world is arranged so that deleting a process also unwinds any RESETxxx expressions that were performed in the process and are still waiting to be unwound, exactly as if a control-D had reset the process to the top. Additionally, there is an implicit RESETLST at the top of each process, so that RESETSAVE can be used as a way of providing "cleanup" functions for when a process is deleted. For these, the value of RESETSTATE is NIL if the process finished normally, ERROR if it was aborted by an error, RESET if the process was explicitly deleted, and HARDRESET if the process is being restarted (after a HARDRESET or a RESTART.PROCESS).

18.20.6 Typein and the TTY Process

There is one global resource, the keyboard, that is particularly problematic to share among processes. Consider, for example, having two processes both performing (READ T). Since the keyboard input routines block while there is no input, both processes would spend most of their time blocking, and it would simply be a matter of chance which process received each character of typein.

To resolve such dilemmas, the system designates a distinguished process, termed the *tty process*, that is assumed to be the process that is involved in terminal interaction. Any typein from the keyboard goes to that process. If a process other than the tty process requests keyboard input, it blocks until it becomes the tty process. When the tty process is switched (in any of the ways described further below), any typeahead that occurred before the switch is saved and associated with the current tty process. Thus, it is always the case the keystrokes are sent to the process that is the tty process at the time of the keystrokes, regardless of when that process actually gets around to reading them.

It is less immediately obvious how to handle keyboard interrupt characters, as their action is asynchronous and not always tied to typein. Interrupt handling is described on page 18.35.

18.20.6.1 Switching the TTY Process

Any process can make itself be the tty process by calling TTY.PROCESS.

(TTY.PROCESS *PROC*) [Function]

Returns the handle of the current tty process. In addition, if *PROC* is non-NIL, makes it be the tty process. The special case of *PROC* = T is interpreted to mean the executive process; this is sometimes useful when a process wants to explicitly give up being the tty process.

(TTY.PROCESSP *PROC*) [Function]

True if *PROC* is the tty process; *PROC* defaults to the running process. Thus, (TTY.PROCESSP) is true if the caller is the tty process.

(WAIT.FOR.TTY) [Function]

Efficiently waits until (TTY.PROCESSP) is true. WAIT.FOR.TTY is called internally by the system functions that read from the terminal; user code thus need only call it in special cases.

In some cases, such as in functions invoked as a result of mouse action or a user's typed-in call, it is reasonable for the function to invoke TTY.PROCESS itself so that it can take subsequent user type in. In other cases, however, this is too undisciplined; it is desirable to let the user designate which process typein should be directed to. This is most conveniently done by mouse action.

The system supports the model that "to type to a process, you click in its window." To cooperate with this model, any process desiring keyboard input should put its process handle as the PROCESS property of its window(s). To handle the common case, the function TTYDISPLAYSTREAM does this automatically when the ttydisplaystream is switched to a new window. A process can own any number of windows; clicking in any of those windows gives the process the tty.

This mechanism suffices for most casual process writers. For example, if a process wants all its input/output interaction to occur in a particular window that it has created, it should just make that window be its tty window by calling TTYDISPLAYSTREAM. Thereafter, it can PRINT or READ to/from the T stream; if the process is not the tty process at the time that it calls READ, it will block until the user clicks in the window.

For those needing tighter control over the tty, the default behavior can be overridden or supplemented. The remainder of this section describes the mechanisms involved.

There is a window property WINDOWENTRYFN that controls whether and how to switch the tty to the process owning a window. The mouse handler, before invoking any normal BUTTONEVENTFN, specifically notices the case of a button going down in a window that belongs to a process (i.e., has a PROCESS window property) that is not the tty process. In this case, it invokes the window's WINDOWENTRYFN of one argument (*WINDOW*). WINDOWENTRYFN defaults to GIVE.TTY.PROCESS:

(GIVE.TTY.PROCESS *WINDOW*) [Function]

If *WINDOW* has a PROCESS property, performs (TTY.PROCESS (WINDOWPROP *WINDOW* 'PROCESS)) and then invokes *WINDOW*'s BUTTONEVENTFN function (or RIGHTBUTTONFN if the right button is down).

There are some cases where clicking in a window does not always imply that the user wants to talk to that window. For example, clicking in a text editor window with a shift key held down means to "shift-select" some piece of text into the input buffer of the *current* tty process. The editor supports this by supplying a WINDOWENTRYFN that performs GIVE.TTY.PROCESS if no shift key is down, but goes into its shift-select mode, without changing the tty process, if a shift key is down. The shift-select mode performs a BKSYSBUF of the selected text when the shift key is let up, the BKSYSBUF feeding input to the current tty process.

Sometimes a process wants to be notified when it becomes the tty process, or stops being the tty process. For example, Chat (page 20.18) turns off all keyboard interrupt characters while it is the tty process, so that they can be passed transparently to the remote host. To support this, there are two process properties, TTYEXITFN and TTYENTRYFN. The actions taken by TTY.PROCESS when it switches the tty to a new process are as follows: the former tty process's TTYEXITFN is called with two arguments (*OLDTTYPROCESS NEWTTYPROCESS*); the new process is made the tty process; finally, the new tty process's TTYENTRYFN is called with two arguments (*NEWTTYPROCESS OLDTTYPROCESS*). Normally the TTYENTRYFN and TTYEXITFN need only their first argument, but the other process involved in the switch is supplied for completeness. In the present system, most processes want to interpret the keyboard in the same way, so it is considered the responsibility of any process that changes the keyboard interpretation to restore it to the normal state by its TTYEXITFN.

A window is "owned" by the last process that anyone gave as the window's PROCESS property. Ordinarily there is no conflict here, as processes tend to own disjoint sets of windows (though, of course, cooperating processes can certainly try to confuse each other). The only likely problem arises with that most global of windows, PROMPTWINDOW. Programs should not be tempted to read from PROMPTWINDOW. This is not usually necessary anyway, as the first attempt to read from T in a process that has not set its TTYDISPLAYSTREAM to its own window causes a tty window to be created for the process (see page

18.32).

18.20.6.2 Handling of Interrupts

At the time that a keyboard interrupt character (page 9.17) is struck, any process could be running, and some decision must be made as to which process to actually interrupt. To the extent that keyboard interrupts are related to typein, most interrupts are taken in the tty process; however, the following are handled specially:

RESET, ERROR
(normally control-D and control-E) These interrupts are taken in the mouse process, if the mouse is not in its idle state; otherwise they are taken in the tty process. Thus, control-E can be used to abort some mouse-invoked window action, such as the Shape command. As a consequence, note that if the mouse invokes some lengthy computation that the user thinks of as "background", control-E still aborts it, even though that may not have been what the user intended. Such lengthy computations, for various reasons, should generally be performed by spawning a separate process to perform them.

The RESET interrupt in a process other than the executive is interpreted exactly as if an error unwound the process to its top level: if the process was designated RESTARTABLE = T, it is restarted; otherwise it is killed.

HELP (Initially control-H) A menu of processes is presented to the user, who is asked to select which one the interrupt should occur in. The current tty process appears with a * next to its name at the top of the menu. The menu also includes an entry "[Spawn Mouse]", for the common case of needing a mouse because the mouse process is currently tied up running someone's BUTTONEVENTFN; selecting this entry spawns a new mouse process, and no break occurs.

BREAK (Initially control-B) Performs the HELP interrupt always in the tty process.

RUBOUT (Initially <del>) This interrupt clears typeahead in *all* processes.

RAID, STACK OVERFLOW, STORAGE FULL
These interrupts always occur in whatever process was running at the time the interrupt struck. In the cases of STACK OVERFLOW and STORAGE FULL, this means that the interrupt is more likely to strike in the offending process (especially if it is a "runaway" process that is not blocking). Note, however, that this process is still not necessarily the guilty party; it could be an innocent bystander that just happened to use up the last of a resource prodigiously consumed by some other process.

18.20.7 Keeping the Mouse Alive

Since the window mouse handler runs in its own process, it is not available while a window's BUTTONEVENTFN function (or any of the other window functions invoked by mouse action) is running. This leads to two sorts of problems: (1) a long computation underneath a BUTTONEVENTFN deprives the user of the mouse for other purposes, and (2) code that runs as a BUTTONEVENTFN cannot rely on other BUTTONEVENTFNs running, which means that there some pieces of code that run differently from normal when run under the mouse process. These problems are addressed by the following functions:

(SPAWN.MOUSE —) [Function]

Spawns another mouse process, allowing the mouse to run even if it is currently "tied up" under the current mouse process. This function is intended mainly to be typed in at the Lisp executive when the user notices the mouse is busy.

(ALLOW.BUTTON.EVENTS) [Function]

Performs a (SPAWN.MOUSE) only when called underneath the mouse process. This should be called (once, on entry) by any function that relies on BUTTONEVENTFNs for completion, if there is any possibility that the function will itself be invoked by a mouse function.

It never hurts, at least logically, to call SPAWN.MOUSE or ALLOW.BUTTON.EVENTS needlessly, as the mouse process arranges to quietly kill itself if it returns from the user's BUTTONEVENTFN and finds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

18.20.8 Debugging Processes

(PROCESS.STATUS.WINDOW *WHERE*) [Function]

Puts up a window that provides several debugging commands for manipulating running processes. If the window is already up, PROCESS.STATUS.WINDOW refreshes it. If *WHERE* is a position, the window is placed in that position; otherwise, the user is prompted for a position.

The window consists of two menus. The first is a menu of all the processes at the moment. Commands in the second menu operate on the process selected in the first menu. The commands are:

BT, BTV, BTV*, BTV!
Performs a backtrace of the selected process. The first time, it prompts for a window in which to display the backtrace.

WHO? Changes the selection to the tty process, i.e., the one currently in control of the keyboard.

KBD← Associates the keyboard with the selected process; i.e., makes the selected process be the tty process.

INFO If the selected process has an INFOHOOK, calls it. The hook may be a function, which is then applied to two arguments, the process and the button (LEFT or MIDDLE) used to invoke INFO, or a form, which is simply EVAL'ed. The APPLY or EVAL happens in the context of the selected process, using PROCESS.APPLY or PROCESS.EVAL. The info hook can be set using PROCESSPROP.

KILL Deletes the selected process.

RESTART
Restarts the selected process.

WAKE Wakes the selected process. Prompts for a value to wake it with (see WAKE.PROCESS).

SUSPEND
Suspends the selected process; i.e., causes it to block indefinitely (until explicitly woken).

BREAK Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.

Currently, the process status window runs under the mouse process, like other menus, so if the mouse is unavailable (e.g., a mouse function is performing an extensive computation), you may be unable to use the process status window (you can try SPAWN.MOUSE, of course).

18.20.9 Non-Process Compatibility

This section describes some considerations for authors of programs that ran in the old single-process Interlisp-D environment, and now want to make sure they run properly in the Multi-processing world. The biggest problem to watch out for is code that runs underneath the mouse handler. Writers of mouse handler functions should remember that in the process world the mouse handler runs in its own process, and hence (a) you cannot depend on finding information on the stack (stash it in the window instead), and (b) while your function is running, the mouse is not available (if you have any non-trivial computation to do, spawn a process to do it, notify one of your existing processes to do it, or use PROCESS.EVAL to run it under some other process).

The following functions are meaningful even if the process world is not on: BLOCK (invokes the system background routine, which includes handling the mouse); TTY.PROCESS, THIS.PROCESS (both return NIL); and TTY.PROCESSP (returns T, i.e., anyone is allowed to take tty input). In addition, the following two functions exist in both worlds:

(EVAL.AS.PROCESS *FORM*) [Function]
Same as (ADD.PROCESS *FORM* 'RESTARTABLE 'NO), when processes are running, EVAL when not. This is highly recommended for mouse functions that perform any non-trivial activity.

(EVAL.IN.TTY.PROCESS *FORM* *WAITFORRESULT*) [Function]
Same as (PROCESS.EVAL (TTY.PROCESS) *FORM* *WAITFORRESULT*), when processes are running, EVAL when not.

Most of the process functions that do not take a process argument can be called even if processes aren't running. ADD.PROCESS creates, but does not run, a new process (it runs when PROCESSWORLD is called).

18.21 PROMPTFORWORD

PROMPTFORWORD is a function that reads in a sequence of characters, generally from the keyboard, without involving READ-like syntax. The intent is to mimic the prompted-read used by the Alto Exec when asking for login names, passwords etc. Thus a user can supply a prompting string, as well as a "candidate" string, which is printed and used if the user types only a word terminator character (or doesn't type anything before a given time limit). As soon as any characters are typed the "candidate" string is erased and the new input takes its place.

PROMPTFORWORD accepts user type-in until one of the "word terminator" characters is typed. Normally, the word terminator characters are EOL, ESCAPE, LF, SPACE, or TAB. This list can be changed using the *TERMINCHAR.LST* argument to PROMPTFORWORD, for example if it is desirable to allow the user to input lines including spaces.

PROMPTFORWORD also recognizes the following special characters:

Control-A, BS, or DEL
: Any of these characters deletes the last character typed and appropriately erases it from the echo stream if it is a displaystream.

Control-W or Control-Q
: Erases all the type-in so far.

Control-R
: Reprints the accumulated string.

?
: Calls up a "help" facility. The action taken is defined by the *GENERATE?LIST.FN* argument to PROMPTFORWORD (see below). Normally, this prints a list of possible candidates.

Control-V
: "Quotes" the next character: after typing Control-V, the next character typed is added to the accumulated string, regardless of any special meaning it has. Allows the user to include editing characters and word terminator characters in the accumulated string.

(PROMPTFORWORD *PROMPT.STR CANDIDATE.STR GENERATE?LIST.FN ECHO.CHANNEL DONTECHOTYPEIN.FLG TIMELIMIT.secs TERMINCHARS.LST KEYBD.CHANNEL OLDSTRING*) [Function]

PROMPTFORWORD has a multiplicity of features, which are specified through a rather large number of input arguments, but the default settings for them (i.e., when they aren't given, or are given as NIL) is such to minimize the number needed in the average case, and an attempt has been made to order the more frequently non-defaulted arguments at the first of the argument list. The default input and echo are both to the terminal; the terminal table in effect during input allows most control characters to be INDICATE'd.

PROMPTFORWORD returns NIL if a null string is typed; this would occur when no candidate is given and only a terminator is typed, or when the candidate is erased and a terminator is typed with no other input still un-erased. In all other cases, PROMPTFORWORD returns a string.

PROMPTFORWORD uses a MONITORLOCK (see page 18.30) so that a second call cannot be started before the first one finished; primarily this is to limit confusion between multiple processes that might try to access the keyboard at the same time, or print in the prompt window "at the same time"

PROMPTFORWORD is controlled through the following arguments:

PROMPT.STR
: If non-NIL, this is coerced to a string and used for prompting; an additional space is output after this string.

CANDIDATE.STR

If non-NIL, this is coerced to a string and offered as initial contents of the input buffer.

GENERATE?LIST.FN

If non-NIL, this is either a string to be printed out for help, or a function to be applied to *PROMPT.STR* and *CANDIDATE.STR* (after both have been coerced to strings), and which should return a list of potential candidates. The help string or list of potential candidates will then be printed on a separate line, the prompt will be restarted, and any type-in will be re-echoed.

Note: If *GENERATE?LIST.FN* is a function, its value list will be "cached" so that it will be run at most once per call to PROMPTFORWORD.

ECHO.CHANNEL

Coerced to an output stream; NIL defaults to T, the "terminal output stream", normally (TTYDISPLAYSTREAM). To achieve echoing to the "current output file", use (GETSTREAM NIL 'OUTPUT). If echo is to a display stream, it will have a flashing caret showing where the next input is to be echoed.

DONTECHOTYPEIN.FLG

If T, there is no echoing of the input characters. If the value of *DONTECHOTYPEIN.FLG* is a single-character atom or string, that character is echoed instead of the actual input. For example, LOGIN prompts for a password with *DONTECHOTYPEIN.FLG* being "*".

TIMELIMIT.secs

If non-NIL, this is the number of seconds (as an integer) that the caller is is willing to wait with no input from *KEYBD.CHANNEL* (see below); if timeout is reached, then *CANDIDATE.WORD* is returned, regardless of any other type-in activity.

TERMINCHAR.LST

This is list of "word terminators"; it defaults to (CHARCODE (EOL ESCAPE LF SPACE TAB)).

KEYBD.CHANNEL

If non-NIL, this is coerced to a stream, and the input bytes are taken from that stream. NIL defaults to the keyboard input stream. Note that this is *not* the same as T, which is a *buffered* keyboard input stream, not suitable for use with PROMPTFORWORD.

OLDSTRING

If non-NIL, this must be a string, which will be destructively used to return the answer.

Examples:

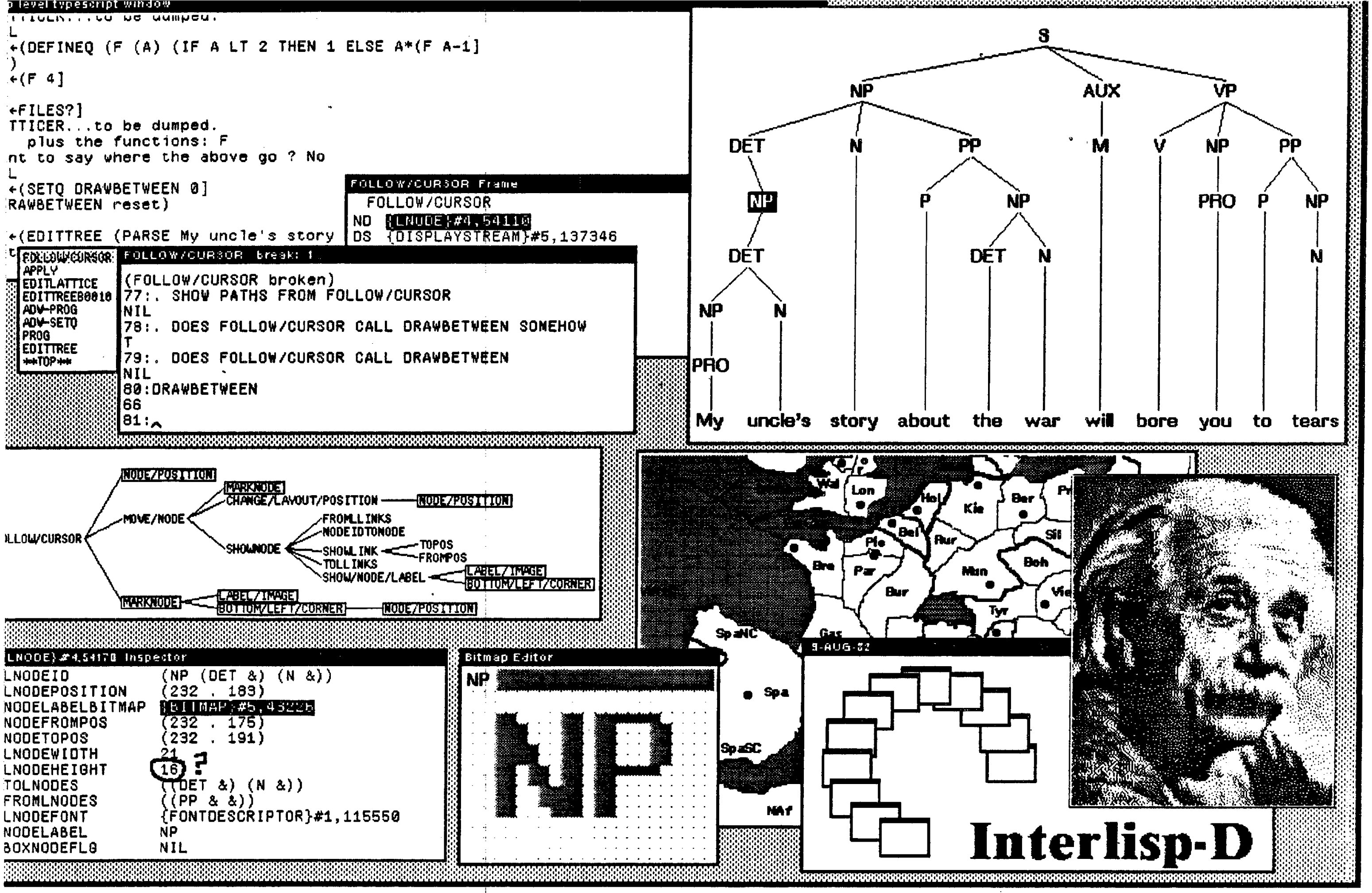
```
(PROMPTFORWORD
   "What is your FOO word?" 'Mumble
   (FUNCTION (LAMBDA () '(Grumble Bletch)))
   PROMPTWINDOW NIL 30)
```

This first prompts the user for input by printing the first argument as a prompt into PROMPTWINDOW; then the proffered default answer, "Mumble", is printed out and the caret starts flashing just after it to indicate that the upcoming input will be echoed there. If the user fails to complete a word within 30 seconds, then the result will be the string "Mumble".

```
(FRESHLINE T)
```

```
(LIST
   (PROMPTFORWORD
      (CONCAT "{"  HOST  "} Login:")
      (USERNAME NIL NIL T))
   (PROMPTFORWORD
      " (password)" NIL NIL NIL '*))
```

This first prompts in whatever window is currently (TTYDISPLAYSTREAM), and then takes in a username; the second call prompts with " (password)" and takes in another word (the password) *without* proffering a candidate, echoing the typed-in characters as "*".

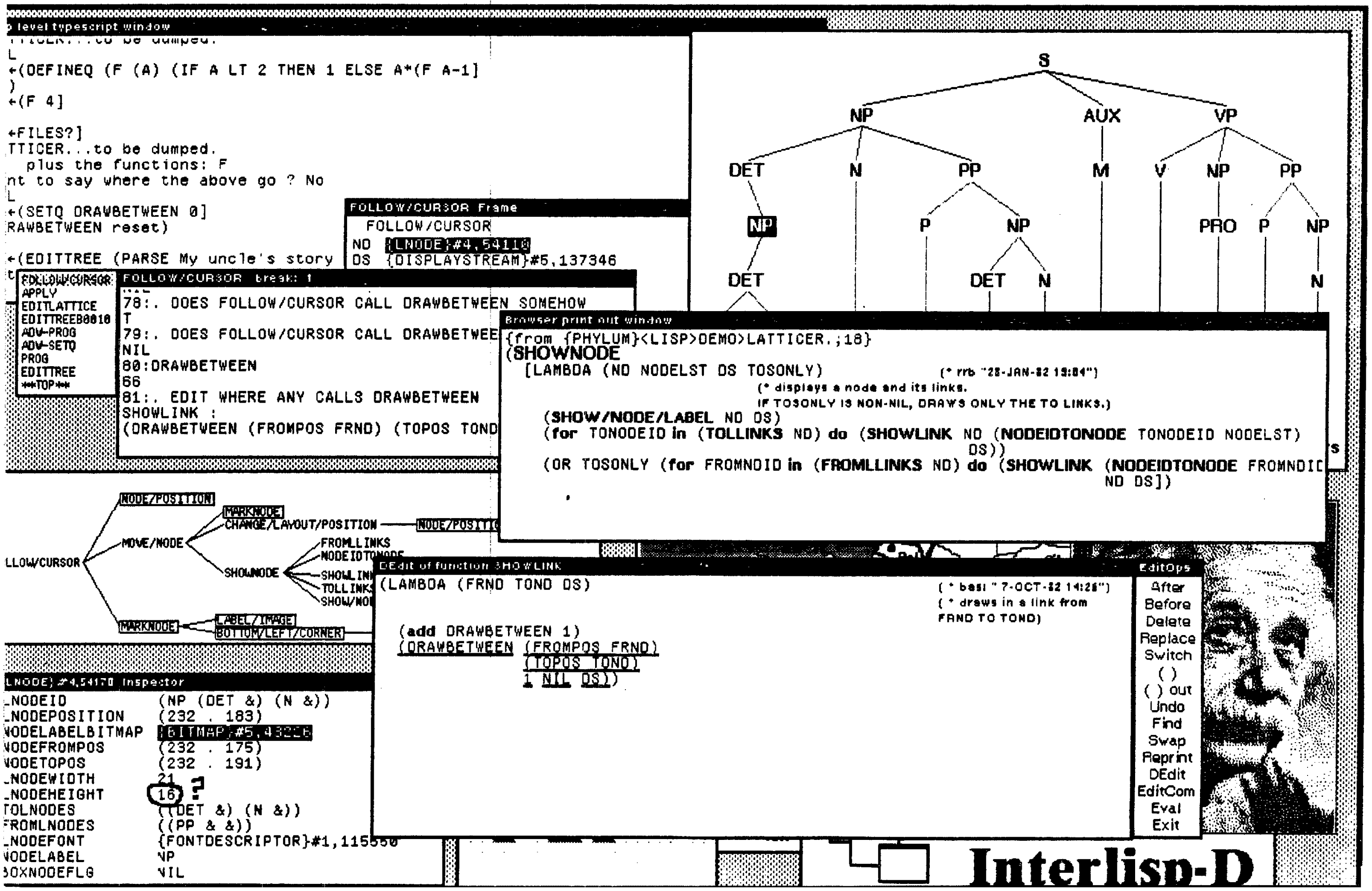


```
p level typescript window
TTICER...to be dumped.
L
←(DEFINEQ (F (A) (IF A LT 2 THEN 1 ELSE A*(F A-1]
)
←(F 4]

←FILES?]
TTICER...to be dumped.
  plus the functions: F
nt to say where the above go ? No
L
←(SETQ DRAWBETWEEN 0]
RAWBETWEEN reset)

←(EDITTREE (PARSE My uncle's story
```

```
FOLLOW/CURSOR Frame
 FOLLOW/CURSOR
ND  {LNODE}#4,54110
DS  {DISPLAYSTREAM}#5,137346
```

```
FOLLOW/CURSOR  break: 1
(FOLLOW/CURSOR broken)
77:. SHOW PATHS FROM FOLLOW/CURSOR
NIL
78:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN SOMEHOW
T
79:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN
NIL
80:DRAWBETWEEN
66
81:
```

```
{LNODE}#4,54170 Inspector
LNODEID          (NP (DET &) (N &))
LNODEPOSITION    (232 . 183)
NODELABELBITMAP  {BITMAP}#5,43226
NODEFROMPOS      (232 . 175)
NODETOPOS        (232 . 191)
LNODEWIDTH       21
LNODEHEIGHT      16 ?
TOLNODES         ((DET &) (N &))
FROMLNODES       ((PP & &))
LNODEFONT        {FONTDESCRIPTOR}#1,115550
NODELABEL        NP
BOXNODEFLG       NIL
```



```
p level typescript window
TTICER...to be dumped.
L
←(DEFINEQ (F (A) (IF A LT 2 THEN 1 ELSE A*(F A-1]
)
←(F 4]

←FILES?]
TTICER...to be dumped.
  plus the functions: F
nt to say where the above go ? No
L
←(SETQ DRAWBETWEEN 0]
RAWBETWEEN reset)

←(EDITTREE (PARSE My uncle's story
```

```
FOLLOW/CURSOR Frame
 FOLLOW/CURSOR
ND  {LNODE}#4,54110
DS  {DISPLAYSTREAM}#5,137346
```

```
FOLLOW/CURSOR  break: 1
78:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN SOMEHOW
T
79:. DOES FOLLOW/CURSOR CALL DRAWBETWEE
NIL
80:DRAWBETWEEN
66
81:. EDIT WHERE ANY CALLS DRAWBETWEEN
SHOWLINK :
(DRAWBETWEEN (FROMPOS FRND) (TOPOS TOND
```

```
Browser print out window
{from {PHYLUM}<LISP>DEMO>LATTICER.;18}
(SHOWNODE
  [LAMBDA (ND NODELST DS TOSONLY)                    (* rrb "28-JAN-82 19:84")
                                        (* displays a node and its links.
                                        IF TOSONLY IS NON-NIL, DRAWS ONLY THE TO LINKS.)
    (SHOW/NODE/LABEL ND DS)
    (for TONODEID in (TOLLINKS ND) do (SHOWLINK ND (NODEIDTONODE TONODEID NODELST)
                                                    DS))
    (OR TOSONLY (for FROMNDID in (FROMLLINKS ND) do (SHOWLINK (NODEIDTONODE FROMNDIC
                                                                  ND DS])
```

```
DEdit of function SHOWLINK
(LAMBDA (FRND TOND DS)                              ( * basi " 7-OCT-82 14:26")
                                                    ( * draws in a link from
                                                    FRND TO TOND)

  (add DRAWBETWEEN 1)
  (DRAWBETWEEN (FROMPOS FRND)
               (TOPOS TOND)
               1 NIL DS))
```

```
{LNODE}#4,54170 Inspector
LNODEID          (NP (DET &) (N &))
LNODEPOSITION    (232 . 183)
NODELABELBITMAP  {BITMAP}#5,43226
NODEFROMPOS      (232 . 175)
NODETOPOS        (232 . 191)
LNODEWIDTH       21
LNODEHEIGHT      16 ?
TOLNODES         ((DET &) (N &))
FROMLNODES       ((PP & &))
LNODEFONT        {FONTDESCRIPTOR}#1,115550
NODELABEL        NP
BOXNODEFLG       NIL
```

CHAPTER 19

INTERLISP-D DISPLAY FACILITIES

This chapter describes the functions that support the display and the interaction with programs that use the display. First, a brief introductory view of using the Interlisp-D display and how some of the other Interlisp facilities have been extended to include display interfaces. The two screen images at left show some of the display features as used by exploratory programming tools of the Interlisp-D environment. The screen is divided into several rectangular areas or windows, each of which provides a view onto some data or process and which can be reshaped and repositioned at will by the user. When they overlap, the occluded portion of the lower window is automatically saved, so that it can be restored when the overlapping window is removed. Since the display is bitmapped, each window can contain an arbitrary mixture of text, lines, curves, and half-tone and solid area images.

The typescript window is in the upper left corner of the screen. It corresponds to the output channel T. In it, the user has defined a program F (factorial) and has then immediately run it, giving an input of 4 and getting a result of 24. Next, he queries the state of his files using the file package function FILES?, finding that one file has been changed (previously) and one function (F) has been defined but not associated with any file yet. The user sets the value of DRAWBETWEEN to 0 in command 74, and the system notes that this is a change and adds DRAWBETWEEN to the set of "changed objects" that might need to be saved.

Then, the user runs his program EDITTREE, giving it a parse tree for the sentence "My uncle's story about the war will bore you to tears". This opens up the big window on the right in which the sentence diagram is drawn. Using the mouse, the user starts to move the NP node on the left (which is inverted to show that it is being moved). While the move is taking place, the user interrupts the tree editor using Control-H, which suspends the computation and causes three "break" windows to appear on top of the lower edge of the typescript. These are part of the window break package. The smallest window shows the dynamic state of the computation, which has been broken inside a subprogram called FOLLOW/CURSOR. The "FOLLOW/CURSOR Frame" window to the right shows the value of the local variables bound by FOLLOW/CURSOR. One of them has been selected (and so appears inverted) and in response, its value has been shown in more detail in the window at the lower left of the screen. The user has marked one of the component values as suspicious by drawing on it using the window command PAINT. In addition, he has asked to examine the contents of the BITMAP component, which used the function EDITBM to open a bitmap edit window to the right. This shows an enlarged copy of the actual NP image that is being moved by the tree editor.

Inside the largest break window, the user has asked some questions about FOLLOW/CURSOR, and queried the value of DRAWBETWEEN (now 66). Using the BROWSER lispusers package, the Masterscope SHOW PATHS command brought up the horizontal tree diagram on the left, which shows which subprograms call each other, starting at FOLLOW/CURSOR. Each node in the call tree produced by the SHOW PATHS command is an active element which will respond to the user's selecting it with the mouse. In the second image, the uscr has selected the SHOWNODE subprogram, which has caused its code to be retrieved from the file (<LISP>DEMO>LATTICER) on the remote file server (PHYLUM) where it was stored and displayed in the "Browser printout window" which has been opened at middle right. User programs and extended Lisp forms (like for and do) are highlighted by system generated font changes. By selecting nodes in the SHOW PATHS window, the user could also have edited or obtained a summary description of any of the subprograms.

Instead, the user told Masterscope (in the break typescript window) to edit wherever anyone calls the DRAWBETWEEN program (a line drawing function). This request causes the system to consult its (dynamically maintained) database of information about user programs, wherein it finds that the subprogram SHOWLINK calls DRAWBETWEEN. It therefore loads the code for SHOWLINK into an edit window which appears under the "Browser print out window". The system then automatically finds and underlines the first (and only) call on DRAWBETWEEN. On the previous line, DRAWBETWEEN is used as a variable (the one the user set and interrogated earlier). The system, however, knows that this is not a subprogram call, so it has been skipped. If the user makes any change to SHOWLINK in the editor, not only will the change take effect immediately, but SHOWLINK will be marked as needing to be updated in its file and the information about it in the program database will be updated. This, in turn, will cause the SHOW PATHS window to be repainted, as its display may no longer be valid.

The Interlisp-D display facility has several layers. At the lowest level are routines which view the display as a collection of bits and provides primitives for moving blocks of bits around (BITBLT). The concepts important to this level are positions, regions and bitmaps. The next level is the display stream, an abstraction that implements clipping to rectangular areas of the screen, line and curve drawing, and printing to the screen in different fonts. The concepts important to this level are fonts and display streams. On the input side, there is a low level interface for reading the display input devices, the cursor location and the mouse buttons. The input and output come together at the next level, the window system which allows areas of the screen used by different programs to overlap by keeping track of information covered and providing control primitives for mouse interaction. This chapter is organized according to these levels.

19.1 POSITION

A position denotes a point in an X,Y coordinate system. A POSITION is an instance of a record with fields XCOORD and YCOORD and is manipulated with the standard record package facilities. For example, (create POSITION XCOORD ← 10 YCOORD ← 20) creates a position representing the point (10,20).

(POSITIONP *x*) [Function]

Returns *x* if *x* is a POSITION; NIL otherwise.

19.2 REGION

A Region denotes a rectangular area in a coordinate system. Regions are characterized by the coordinates of their bottom left corner and their width and height. A REGION is a record with fields LEFT, BOTTOM, WIDTH, and HEIGHT. It can be manipulated with the standard record package facilities. There are access functions for the REGION record that returns the TOP and RIGHT of the region.

The following functions are provided for manipulating regions:

(CREATEREGION *LEFT BOTTOM WIDTH HEIGHT*) [Function]

Returns an instance of the REGION record which has *LEFT*, *BOTTOM*, *WIDTH* and

HEIGHT as respectively its LEFT, BOTTOM, WIDTH, and HEIGHT.

Example: (CREATEREGION 10 -20 100 200) will create a region that denotes a rectangle whose width is 100, whose height is 200, and whose lower left corner is (10,-20).

(INTERSECTREGIONS *REGION*$_1$ *REGION*$_2$ ··· *REGION*$_n$) [NoSpread Function]

Returns a region which is the intersection of a number of regions. Returns NIL if the intersection is empty. If there are no regions given, it returns a very large region.

(UNIONREGIONS *REGION*$_1$ *REGION*$_2$ ··· *REGION*$_n$) [NoSpread Function]

Returns a region which is the union of a number of regions, i.e. the smallest region that contains all of them. Returns NIL if there are no regions given.

(REGIONSINTERSECTP *REGION1* *REGION2*) [Function]

Returns T if *REGION1* intersects *REGION2*. Returns NIL if they do not intersect.

(SUBREGIONP *LARGEREGION* *SMALLREGION*) [Function]

Returns T if *SMALLREGION* is a subregion (is equal to or entirely contained in) *LARGEREGION*; otherwise returns NIL.

(EXTENDREGION *REGION* *INCLUDEREGION*) [Function]

Changes (destructively modifies) the region *REGION* so that it includes the region *INCLUDEREGION*. It returns *REGION*.

(INSIDEP *REGION* *X* *Y*) [Function]

If *X* and *Y* are numbers, it returns T if the point (*X*,*Y*) is inside of *REGION*. If *X* is a POSITION, it returns T if *X* is inside of *REGION*. Otherwise, it returns NIL.

19.3 BITMAP

The display primitives manipulate graphical images in the form of bitmaps. A bitmap is a rectangular array of "pixels," each of which is an integer representing the color of one point in the bitmap image. A bitmap is created with a specific number of bits allocated for each pixel. Most bitmaps used for the display screen use one bit per pixel, so that at most two colors can be represented. If a pixel is 0, the corresponding location on the image is white. If a pixel is 1, its location is black. (This interpretation can be changed with the function VIDEOCOLOR; see page 19.7.) Bitmaps with more than one bit per pixel are used to represent color or grey scale images.

Bitmaps use a positive integer coordinate system with the lower left corner pixel at coordinate (0,0). Bitmaps are represented as instances of the datatype BITMAP with fields BITMAPWIDTH, BITMAPHEIGHT, BITMAPBITSPERPIXEL, BITMAPRASTERWIDTH, and BITMAPBASE. Only the width, height, and bits per pixel fields are of interest to the user, and can be accessed with the following functions:

(BITMAPWIDTH *BITMAP*) [Function]

Returns the width of *BITMAP* in pixels.

(BITMAPHEIGHT *BITMAP*) [Function]

Returns the height of *BITMAP* in pixels.

(BITSPERPIXEL *BITMAP*) [Function]

Returns the number of bits per pixel of *BITMAP*.

The functions used to manipulate bitmaps are:

(BITMAPCREATE *WIDTH HEIGHT BITSPERPIXEL*) [Function]

Creates and returns a new bitmap which is *WIDTH* pixels wide by *HEIGHT* pixels high, with *BITSPERPIXEL* pits per pixel. If *BITSPERPIXEL* is NIL, the default is 1.

(BITMAPBIT *BITMAP X Y NEWVALUE*) [Function]

If *NEWVALUE* is between 0 and the maximum value for a pixel in *BITMAP*, the pixel (*X*,*Y*) is changed to *NEWVALUE* and the old value is returned. If *NEWVALUE* is NIL, *BITMAP* is not changed but the value of the pixel is returned. If *NEWVALUE* is anything else, an error is generated. If (*X*,*Y*) is outside the limits of *BITMAP*, 0 is returned and no pixels are changed. *BITMAP* can also be a window.

(BITMAPCOPY *BITMAP*) [Function]

Returns a new bitmap which is a copy of *BITMAP* (same dimensions and contents).

(EXPANDBITMAP *BITMAP WIDTHFACTOR HEIGHTFACTOR*) [Function]

Returns a new bitmap that is *WIDTHFACTOR* times as wide as *BITMAP* and *HEIGHTFACTOR* times as high. Each pixel of *BITMAP* is copied into a *WIDTHFACTOR* times *HEIGHTFACTOR* block of pixels. If NIL, *WIDTHFACTOR* defaults to 4, *HEIGHTFACTOR* to 1.

There are two distinguished bitmaps that are read by the hardware to become visible as the screen and the cursor. The screen is a bitmap SCREENWIDTH (=1024) wide by SCREENHEIGHT (=808) high. The cursor is a bitmap CURSORWIDTH (=16) wide by CURSORHEIGHT (=16) high. They are accessed by:

(SCREENBITMAP) [Function]

Returns the screen bitmap.

(CURSORBITMAP) [Function]

Returns the cursor bitmap.

Note: The cursor bitmap can be changed with the function CURSOR (page 19.16).

19.4 BITBLT

BITBLT is the primitive function for moving bits from one bitmap to another. It is similar to the function RASTEROP that is used in other systems.

(BITBLT *SOURCEBITMAP SOURCELEFT SOURCEBOTTOM DESTINATIONBITMAP DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION*) [Function]

WIDTH and *HEIGHT* define a pair of rectangles, one in each of the *SOURCEBITMAP* and *DESTINATIONBITMAP* whose left, bottom corners are at, respectively, (*SOURCELEFT*, *SOURCEBOTTOM*) and (*DESTINATIONLEFT*, *DESTINATIONBOTTOM*). If these rectangles overlap the boundaries of either bitmap they are both reduced in size (without translation) so that they fit within their respective boundaries. If *CLIPPINGREGION* is non-NIL it should be a REGION and is interpreted as a clipping region within *DESTINATIONBITMAP*; clipping to this region may further reduce the defining rectangles. These (possibly reduced) rectangles define the source and destination rectangles for BITBLT. *SOURCEBITMAP* and *DESTINATIONBITMAP* can also be display streams or windows, in which case their associated bitmaps are used.

The mode of transferring bits is defined by *SOURCETYPE* and *OPERATION*. *SOURCETYPE* and *OPERATION* specify boolean functions that are used to determine, respectively, the method of combining *SOURCEBITMAP* bits with the *TEXTURE* and the operation between these resultant bits and *DESTINATIONBITMAP*. *TEXTURE* is a gray pattern, as described on page 19.6. (Note: The alignment of the texture pattern with BITBLT is such that the origin of the destination bitmap is at an intersection of the "tiles.")

SOURCETYPE specifies how to combine the bits from *SOURCEBITMAP* with the bits from *TEXTURE* (a background pattern) to produce a "Source". This is designed to allow characters and figures to be placed on a background.

SOURCETYPE	Source
INPUT	*SOURCEBITMAP*
INVERT	(NOT *SOURCEBITMAP*)
TEXTURE	*TEXTURE*

For the INPUT and INVERT case, the *TEXTURE* argument to BITBLT is ignored. For the TEXTURE case, the *SOURCEBITMAP*, *SOURCELEFT*, and *SOURCEBOTTOM* arguments are ignored.

OPERATION specifies how this source is combined with the bits in *DESTINATIONBITMAP* and stored back into *DESTINATIONBITMAP*.

OPERATION	*DESTINATIONBITMAP* becomes
REPLACE	Source
PAINT	(OR *DESTINATIONBITMAP* Source)
INVERT	(XOR *DESTINATIONBITMAP* Source)
ERASE	(AND *DESTINATIONBITMAP* (NOT Source))

SOURCELEFT, *SOURCEBOTTOM*, *DESTINATIONLEFT*, and *DESTINATIONBOTTOM* default to 0. *WIDTH* and *HEIGHT* default to the width and height of the *SOURCEBITMAP*. *TEXTURE* defaults to white. *SOURCETYPE* defaults to INPUT. *OPERATION* defaults to REPLACE. If *CLIPPINGREGION* is not provided, no additional clipping is done. BITBLT returns T if any bits were moved; NIL otherwise.

Note: BITBLT and BITMAPBIT accept windows and display streams as their bitmap arguments. In these cases, the remaining arguments are interpreted as values in the coordinate system of the window or display stream and the operation of the functions are translated and clipped accordingly. If a window or display stream is used as the destination to BITBLT, its clipping region limits the operation involved.

19.5 TEXTURE

A Texture denotes a pattern of gray which can be used by BITBLT to (conceptually) tessellate the plane to form an infinite sheet of gray. It is currently a 4 by 4 pattern. Textures are created interactively using the function EDITSHADE or from bitmaps using the following function.

(CREATETEXTUREFROMBITMAP *BITMAP*) [Function]

Returns a texture object that will produce the texture of *BITMAP*. If *BITMAP* is too large, its lower left portion is used. If *BITMAP* is too small, it is repeated to fill out the texture.

(TEXTUREP *OBJECT*) [Function]

Returns *OBJECT* if it is a texture, i.e. a legal texture argument to BITBLT.

The common textures white and black are available as system constants WHITESHADE and BLACKSHADE. The global variable GRAYSHADE is used by many system facilities as a background gray shade and can be set by the user. The original background shade of the window system is kept in WINDOWBACKGROUNDSHADE. The background shade can be changed by the following function:

(CHANGEBACKGROUND *SHADE*) [Function]

Changes the background shade of the window system. *SHADE* determines the pattern of the background. If *SHADE* is a texture, then the background is simply painted with it. If *SHADE* is a BITMAP, the background is tesselated (tiled) with it to cover the screen. If *SHADE* is T, it changes to the original shade, the value of WINDOWBACKGROUNDSHADE. It returns the previous value of the background.

19.6 SAVING BITMAPS

Bitmaps can be saved on files with the VARS file package command (page 11.22). The following two functions translate bitmaps into and out of a representation which may be used to transfer bitmaps between Interlisp and other computer systems' representations.

(READBITMAP) [Function]

Creates a bitmap by reading an expression (written by PRINTBITMAP) from the primary input channel.

(PRINTBITMAP *BITMAP*) [Function]

Prints the bitmap *BITMAP* on the primary output channel in a format that can be read back in by READBITMAP.

19.7 SCREEN OPERATION

The following functions control the display screen.

(VIDEOCOLOR *BLACKFLG*) [NoSpread Function]

Sets the interpretation of the bits in the screen bitmap. If *BLACKFLG* is NIL, a 0 bit will be displayed as white, otherwise a 0 bit will be displayed as black. VIDEOCOLOR returns the previous setting. If *BLACKFLG* is not given, VIDEOCOLOR will return the current setting without changing anything.

Note: This function only works on the Xerox 1100 and Xerox 1108.

(VIDEORATE *TYPE*) [Function]

Sets the rate at which the screen is refreshed. *TYPE* is one of NORMAL or TAPE. If *TYPE* is TAPE, the screen will be refreshed at the same rate as TV (60 cycles per second). This makes the picture look better when video taping the screen. Note: Changing the rate may change the dimensions of the display on the picture tube.

Several functions are provided for turning off the display (partially or completely). See page 18.22.

19.8 CHARACTERS AND FONTS

Fonts control the way characters look when printed on the screen or a graphics printer. Fonts are defined by a distinctive style or FAMILY (such as Gacha or TimesRoman), a SIZE (such as 10 points), and FACE (such as bold or italic). Fonts also have a ROTATION that indicates the orientation of characters on the screen or page. A normal horizontal font (also called a portrait font) has a rotation of 0; the rotation of a vertical (landscape) font is 90 degrees. While the specification allows any combination, in practice the user will find that only certain combinations of families, sizes, faces, and rotations are available.

In specifying a font to the functions described below, a FAMILY is represented by a literal atom, a SIZE by a positive integer, and a FACE by a three-element list of the form (WEIGHT SLOPE EXPANSION). WEIGHT, which indicates the thickness of the characters, can be BOLD, MEDIUM, or LIGHT; SLOPE can be ITALIC or REGULAR; and EXPANSION can be REGULAR, COMPRESSED, or EXPANDED, indicating how spread out the characters are. For convenience, faces may also be specified by three-character atoms, where each character is the first letter of the corresponding field. Thus, MRR is a synonym for (MEDIUM REGULAR REGULAR). In addition, certain common face combinations may be indicated by special literal atoms:

STANDARD = (MEDIUM REGULAR REGULAR) = MRR

ITALIC = (MEDIUM ITALIC REGULAR) = MIR

BOLD = (BOLD REGULAR REGULAR) = BRR

BOLDITALIC = (BOLD ITALIC REGULAR) = BIR

A font also has the properties ASCENT, DESCENT, and HEIGHT (= ASCENT + DESCENT), and, for each character, a width and bit pattern. The ASCENT is the maximum height of any character in the font from its base line (the printing position). The DESCENT is the maximum extent of any character below the base line, such as the lower part of a "p." Therefore the top line of a character will be at Base+ASCENT-1, while the bottom line will be at Base-DESCENT. The width of each character specifies how a stream's position will change when the character is printed. This may have both an X and a Y component (e.g., for landscape fonts), and it varies from character to character in variable pitch fonts.

The information about a particular font is represented in a font descriptor. The following functions manipulate font descriptors:

(FONTCREATE *FAMILY SIZE FACE ROTATION DEVICE NOERRORFLG*) [Function]

Returns a font descriptor for the specified font. *SIZE* is an integer indicating the size of the font in points. *FACE* specifies the face characteristics in one of the formats listed above; if *FACE* is NIL, STANDARD is used. *ROTATION*, which specifies the orientation of the font, is 0 (or NIL) for a portrait font and 90 for a landscape font. *DEVICE* indicates the output device for the font. For Interlisp-D, the possible values for *DEVICE* are DISPLAY for the display screen and PRESS for Press printers. *DEVICE* defaults to DISPLAY.

For display fonts, FONTCREATE looks for a STRIKE file with the appropriate name (such as TIMESROMAN8BI.STRIKE for a TIMESROMAN 8 BOLDITALIC font), searching through directories on the list FONTDIRECTORIES. If the file is found, it is read into a font descriptor. If the file is not found, FONTCREATE looks for fonts with less face information (in this example, TIMESROMAN8I.STRIKE) and fakes the remaining faces (such as by doubling the bit pattern of each character or slanting it). If no appropriately sized font is found, the action of the function is determined by *NOERRORFLG*. If *NOERRORFLG* is NIL, it generates a FILE NOT FOUND error with the name of the most specific file tried (in the example TIMESROMAN8BI.STRIKE); otherwise, FONTCREATE returns NIL.

For Press fonts, FONTCREATE accesses the widths information for the font from a font-dictionary file whose name is in the list FONTWIDTHSFILES (usually initialized in the site-greeting file to contain at least {DSK}FONTS.WIDTHS). That dictionary must contain information for the face as specified; there is no acceptable faking algorithm for hard-copy fonts. The width and height information for press fonts is expressed in micas (= 10 microns = 1/2540 inch), not in screen-point units.

The *FAMILY* argument to FONTCREATE may also be a list, in which case it is interpreted as a *FAMILY-SIZE-FACE-ROTATION* quadruple. Thus, (FONTCREATE '(GACHA 10 BOLD)) is equivalent to (FONTCREATE 'GACHA 10 'BOLD). *FAMILY* may also be a font descriptor, in which case that descriptor is simply returned.

(FONTP *X*) [Function]

Returns *X* if *X* is a font descriptor; NIL otherwise.

The following functions take a font as one argument. This argument must either be a particular font descriptor or coerceable to a font descriptor. A display stream is coerced to its current font, a window is coerced to the current font of its display stream, and anything else is coerced by applying FONTCREATE to it.

(FONTPROP *FONT PROP*) [Function]

Returns the value of the *PROP* property of font *FONT*. *PROP* may be one of FAMILY, SIZE, FACE, WEIGHT, SLOPE, EXPANSION, DEVICE, ASCENT, DESCENT, HEIGHT, or ROTATION.

(FONTCOPY *OLDFONT* $PROP_1$ VAL_1 $PROP_2$ VAL_2 ···) [NoSpread Function]

Returns a font descriptor that is a copy of the font *OLDFONT*, but which differs from *OLDFONT* in that *OLDFONT*'s properties are replaced by the specified properties and values. Thus, (FONTCOPY *FONT* 'WEIGHT 'BOLD 'DEVICE 'PRESS) will return a bold press font with all other properties the same as those of *FONT*. FONTCOPY accepts all the properties that FONTPROP interrogates except for ASCENT, DESCENT, and HEIGHT. If the first property is a list, it is taken to be the *PROP*$_1$ *VAL*$_1$ *PROP*$_2$ *VAL*$_2$ ··· sequence. Thus, (FONTCOPY *FONT* '(WEIGHT BOLD DEVICE PRESS)) is equivalent to the example above.

(CHARWIDTH *CHARCODE FONT*) [Function]

CHARCODE is an integer that represents a valid character (as returned by CHCON1). Returns the amount by which a stream's X-position will be incremented when the character is printed.

(CHARWIDTHY *CHARCODE FONT*) [Function]

Like CHARWIDTH, but returns the Y component of the character's width, the amount by which a stream's Y-position will be incremented when the character is printed. This will be zero for most characters in normal portrait fonts, but may be non-zero for landscape fonts or for vector-drawing fonts.

(STRINGWIDTH *STR FONT PRIN2FLG RDTBL*) [Function]

Returns the amount by which a stream's X-position will be incremented if the printname for the Interlisp-D object *STR* is printed in font *FONT*. If *FONT* is a display stream, its font is used. If *PRIN2FLG* is non-NIL, the PRIN2-pname of *STR* with respect to the readtable *RDTBL* is used.

(STRINGREGION *STR WINDOW PRIN2FLG RDTBL*) [Function]

Returns the region occupied by *STR* if it were printed at the current location in *WINDOW*. This is useful for determining where text is in a window to allow the user to select it. The arguments *PRIN2FLG* and *RDTBL* are passed to STRINGWIDTH.

It is sometimes useful to simulate an unavailable font or to use a font with characteristics different from the interpretations provided by the system. The following function allows the user to tell the system what font descriptor to use for given characteristics.

(SETFONTDESCRIPTOR *FAMILY SIZE FACE ROTATION DEVICE FONT*) [Function]

Indicates to the system that *FONT* is the font with the *FAMILY SIZE FACE ROTATION DEVICE* characteristics. If *FONT* is NIL, the font associated with these characteristics is cleared and will be recreated the next time it is needed. As with FONTPROP and FONTCOPY, *FONT* is coerced to a font descriptor if it is not one already.

(DEFAULTFONT *DEVICE FONT* —) [Function]

Returns the font that would be used as the default (if NIL were specified as a font argument) for device *DEVICE*. If *FONT* is a font descriptor, it is set to be the default font for *DEVICE*.

The following functions allow the user to access and change the bitmaps for individual characters in a display font.

(GETCHARBITMAP *CHARCODE FONT*) [Function]

Returns a bitmap containing a copy of the image of the character *CHARCODE* in the font *FONT*.

(PUTCHARBITMAP *CHARCODE FONT NEWCHARBITMAP*) [Function]

Changes the bitmap image of the character *CHARCODE* in the font *FONT* to the bitmap *NEWCHARBITMAP*. Currently, *NEWCHARBITMAP* must be the same width and height as the current image for *CHARCODE* in the font *FONT*.

Users can interactively edit characters using the EDITCHAR function (page 20.10).

19.9 DISPLAY STREAMS

Streams are used as the basis for all I/O operations. Files are implemented as streams that can support character printing and reading operations, and file pointer manipulation. Display streams are a type of stream that also provides an interface for translation, clipping, and figure generation on bitmaps. All of the operations that can applied to streams can be applied to display streams. For example, a display stream can be passed as the argument to PRINT, to print something on the bitmap of a display stream. In addition, special functions are provided to draw lines and curves and perform other graphical operations on display streams. Calling these functions on a stream that is not a display stream will generate an error.

Windows are closely related to display streams and can be thought of as a type of display stream. (In the near future, windows will be a type of display stream.) All of the functions that operate on display streams also accept windows.

Display streams can be created with the following function:

(DSPCREATE *DESTINATION*) [Function]

Returns a display stream, with initial settings as indicated below. If *DESTINATION* is specified, it is used as the destination bitmap, otherwise the screen bitmap is used.

Each window has an associated display stream. To get the window of a particular display stream, use:

(WFROMDS *DISPLAYSTREAM*) [Function]

Returns the window associated with *DISPLAYSTREAM*, creating a window if one does not exist. Returns NIL if the destination of *DISPLAYSTREAM* is not a screen bitmap that supports a window system.

19.9.1 Manipulating Display Streams

The following functions manipulate the fields of a display stream (they may also be given a window, in which case the associated display stream is used). These functions return the old value (the one being replaced). A value of NIL for the new value will return the current setting without changing it. These functions do not change any of the bits in the display stream's destination bitmap; just the effect of future operations done through the display stream.

Warning: The window system maintains the Destination, XOffset, YOffset, and ClippingRegion fields of each window's display stream, adjusting them during window operations. Users should be very careful about changing these fields in a window's display stream (with DSPDESTINATION, DSPXOFFSET, DSPYOFFSET, or DSPCLIPPINGREGION).

(DSPDESTINATION *DESTINATION DISPLAYSTREAM*) [Function]

Destination: The bitmap that the display stream modifies. This can be either the screen bitmap, or an auxilliary bitmap in order to construct figures, possibly save them, and then display them in a single operation. Initially the screen bitmap.

(DSPXOFFSET *XOFFSET DISPLAYSTREAM*) [Function]

(DSPYOFFSET *YOFFSET DISPLAYSTREAM*) [Function]

XOffset: The X origin of the display stream's coordinate system in the destination bitmap's coordinate system. Initially 0 (no X-coordinate translation).

YOffset: The Y origin of the display stream's coordinate system in the destination bitmap's coordinate system. Initially 0 (no Y-coordinate translation).

Display streams have their own coordinate system. Having the coordinate system local to the display stream allows objects to be displayed at different places by translating the display stream's coordinate system relative to its destination bitmap.

(DSPCLIPPINGREGION *REGION DISPLAYSTREAM*) [Function]

ClippingRegion: A region that limits the extent of characters printed and lines drawn (in the display stream's coordinate system). Initially set so that no clipping occurs.

(DSPXPOSITION *XPOSITION DISPLAYSTREAM*) [Function]

(DSPYPOSITION *YPOSITION DISPLAYSTREAM*) [Function]

XPosition: The current X position. Initially 0.

YPosition: The current Y position. Initially 0.

DSPXPOSITION and DSPYPOSITION specify the "current position" of the display stream, the position (in the display stream's coordinate system) where the next printing operation will start from. The functions which print characters or draw on a display stream update these values appropriately.

(DSPTEXTURE *TEXTURE DISPLAYSTREAM*) [Function]

Texture: A texture that is the background pattern used for the display stream. Initially the value of WHITESHADE.

(DSPFONT *FONT DISPLAYSTREAM*) [Function]

Font: A Font Descriptor that specifies the font used when printing characters to the display stream. Initially Gacha 10.

Note: DSPFONT determines its new font descriptor from *FONT* by the same coercion rules that FONTPROP and FONTCOPY use, with one additional possibility: If *FONT* is a list of the form (*PROP*$_1$ *VAL*$_1$ *PROP*$_2$ *VAL*$_2$ ···) where *PROP*$_1$ is acceptable as a font-property to FONTCOPY, then the new font is obtained by (FONTCOPY (DSPFONT NIL DISPLAYSTREAM) *PROP*$_1$ *VAL*$_1$ *PROP*$_2$ *VAL*$_2$ ···).

(DSPLEFTMARGIN *XPOSITION DISPLAYSTREAM*) [Function]

LeftMargin: An integer that is the X position after an end-of-line (in the display stream's coordinate system) - initially 0.

(DSPRIGHTMARGIN *XPOSITION DISPLAYSTREAM*) [Function]

RightMargin: An integer that is the maximum X position that characters will be printed at (in the display stream's coordinate system) - initially the value of SCREENWIDTH. This determines when an end of line is automatically inserted by the printing functions.

The line length of a window or display stream (as returned by LINELENGTH, page 6.8) is computed by dividing the distance between the left and right margins by the width of an uppercase "A" in the current font. The line length is changed whenever the Font, LeftMargin, or RightMargin are changed.

(DSPSOURCETYPE *SOURCETYPE DISPLAYSTREAM*) [Function]

SourceType: The BITBLT sourcetype used when printing characters to the display stream. Must be either INPUT or INVERT. Initially INPUT.

(DSPOPERATION *OPERATION DISPLAYSTREAM*) [Function]

Operation: The default BITBLT operation (REPLACE, PAINT, INVERT, or ERASE) used when printing or drawing on the display stream. Initially REPLACE.

(DSPLINEFEED *DELTAY DISPLAYSTREAM*) [Function]

LineFeed: An integer that specifies the Y increment for each linefeed, normally negative. Initially minus the height of the initial font (Gacha 10).

(DSPSCROLL *SWITCHSETTING DISPLAYSTREAM*) [Function]

Scroll: A flag that determines the scrolling behavior of the display stream; either ON or OFF. If ON, the bits in the display streams's destination are moved after any linefeed that moves the current position out of the destination bitmap. Any bits moved out of the current clipping region are lost. Does not adjust the XOffset, YOffset, or ClippingRegion fields. Initially OFF. (Note: if *SWITCHSETTING* is NIL, the Scroll field is *not* changed, and the previous value is returned.)

19.9.2 Drawing on Windows and Display Streams

(DSPFILL *REGION TEXTURE OPERATION DISPLAYSTREAM*) [Function]

Fills *REGION* of the destination bitmap (within the clipping region) with *TEXTURE* (a pattern of bits). If *REGION* is NIL, the whole destination (within the clipping region) is used. If *TEXTURE* or *OPERATION* are NIL, the values from *DISPLAYSTREAM* are used.

(FILLCIRCLE *X Y RADIUS TEXTURE DISPLAYSTREAM*) [Function]

Fills in a circular area of radius *RADIUS* about the point (*X*,*Y*) in the destination bitmap of *DISPLAYSTREAM* with *TEXTURE*. *DISPLAYSTREAM*'s position is left at (*X*,*Y*).

(DSPRESET *DISPLAYSTREAM*) [Function]

Sets the X position of *DISPLAYSTREAM* to its left margin, sets its Y position to the top of the clipping region minus the font ascent, and fills its destination bitmap with its background Texture.

(MOVETO *X Y DISPLAYSTREAM*) [Function]

Changes the current position of *DISPLAYSTREAM* to the point (*X*, *Y*).

(RELMOVETO *DX DY DISPLAYSTREAM*) [Function]

Changes the current position to the point (*DX*, *DY*) coordinates away from current position of *DISPLAYSTREAM*.

(MOVETOUPPERLEFT *DISPLAYSTREAM REGION*) [Function]

Changes the X position to the left edge of *REGION* and the Y position to the top of *REGION* less the font height of *DISPLAYSTREAM*. This is the beginning position of the top line of text in this region. If *REGION* is NIL, the clipping region of *DISPLAYSTREAM* is used. Note: this does not set the X position to the left margin like the function DSPRESET does.

(DSPBACKUP *WIDTH DISPLAYSTREAM*) [Function]

Backs up *DISPLAYSTREAM* over a character which is *WIDTH* screen points wide. DSPBACKUP fills the backed over area with the display stream's background texture and decreases the X position by *WIDTH*. If this would put the X position less than *DISPLAYSTREAM*'s left margin, its operation is stopped at the left margin. It returns T if any bits were written, NIL otherwise.

(CENTERPRINTINREGION *EXP REGION DISPLAYSTREAM*) [Function]

Prints *EXP* so that is it centered within *REGION* of the *DISPLAYSTREAM*. If *REGION* is NIL, *EXP* will be centered in the clipping region of *DISPLAYSTREAM*.

19.9.3 Drawing Lines and Curves

Interlisp-D provides several functions for drawing lines and curves onto the destination bitmap of a display stream or window. The curve drawing functions take their BITBLT operation from the display stream, while for straight lines the Operation may be specified as an argument to the drawing function, with the display stream's operation only being used by default.

The following functions produce straight lines of the specified width (in screen points; the default is 1) in the display stream's destination bitmap. They do not allow "brush" patterns; however, they do support INVERT mode inwhich redrawing a line will erase it. These functions are intended for interactive applications where efficiency is important. DRAWCURVE can be used to draw lines with brushes.

(DRAWTO *X Y WIDTH OPERATION DISPLAYSTREAM COLOR*) [Function]

Draws a line from the current position to the point (*X*, *Y*) onto the destination bitmap of *DISPLAYSTREAM*. The position of *DISPLAYSTREAM* is set to (*X*, *Y*).

If the destination bitmap has multiple bits per pixel, *COLOR* is a color specification that determines the color used to draw the line (See page 19.44). If *COLOR* is NIL, this will be the DSPCOLOR of *DISPLAYSTREAM*.

(RELDRAWTO *DX DY WIDTH OPERATION DISPLAYSTREAM COLOR*) [Function]

Draws a line from the current position to the point (*DX*,*DY*) coordinates away onto the destination bitmap of *DISPLAYSTREAM*. The position of *DISPLAYSTREAM* is set to the end of the line.

(DRAWLINE X_1 Y_1 X_2 Y_2 *WIDTH OPERATION DISPLAYSTREAM COLOR*) [Function]

Draws a line from the point (X_1, Y_1) to the point (X_2, Y_2) onto the destination bitmap of *DISPLAYSTREAM*. The position of *DISPLAYSTREAM* is set to (X_2, Y_2).

(DRAWBETWEEN $POSITION_1$ $POSITION_2$ *WIDTH OPERATION DISPLAYSTREAM COLOR*) [Function]

Draws a line from the point $POSITION_1$ to the point $POSITION_2$ onto the destination bitmap of *DISPLAYSTREAM*. The position of *DISPLAYSTREAM* is set to *POSITION*$_2$.

A curve is drawn by placing a brush pattern centered at each point along the curve's trajectory. A brush pattern is defined by its shape, size, and color. The currently recognized shapes are ROUND, SQUARE, HORIZONTAL, VERTICAL, and DIAGONAL. A brush size is an integer specifying the width of the brush in screen points. The color is a color specification (see page 19.44), which is only used if the curve is drawn on a multiple bits per pixel bitmap.

A brush is specified to the various drawing functions as a shape-width-color list (such as (SQUARE 2) or (VERTICAL 4 RED)). A brush can also be specified as a positive integer, which is interpreted as a ROUND brush of that width. Finally, if a brush is specified as NIL, a (ROUND 1) brush is used as default.

If a brush is a litatom, it is assumed to be a function which is called at each point of the curve's trajectory with three arguments: the X-coordinate or the point, the Y-coordinate, and the display stream.

The appearance of a curve is also determined by its dashing characteristics. Dashing is specified by a list of positive integers. If a curve is dashed, the brush is placed along the trajectory for the number of points indicated by the first element of the dashing list. The brush is *off*, not placed in the bitmap, for a number of points indicated by the second element. The third element indicates how long it will be on again, and so forth. The dashing sequence is repeated from the beginning when the list is exhausted. A curve or line is not dashed if the dashing argument to the drawing function is NIL.

The curve functions use the display stream's clipping region and operation. Because of the problem of overlapping brush points, the REPLACE and INVERT operations are not implemented.

(DRAWCURVE *KNOTS CLOSED BRUSH DASHING DISPLAYSTREAM*) [Function]

Draws a spline curve. *KNOTS* is a list of positions to which the spline will be fitted. *CLOSED* is a flag which indicates whether or not the spline is to be closed. The other arguments are interpreted as described above.

(DRAWCIRCLE *X Y RADIUS BRUSH DASHING DISPLAYSTREAM*) [Function]

Draws a circle of radius *RADIUS* about the point (*X*, *Y*) onto the destination bitmap of *DISPLAYSTREAM*. *DISPLAYSTREAM*'s position is left at (*X*, *Y*). (Dashing may not be implemented for this function yet.) The other arguments are interpreted as described above.

(DRAWELLIPSE *X Y SEMIMINORRADIUS SEMIMAJORRADIUS ORIENTATION BRUSH DASHING DISPLAYSTREAM*) [Function]

Draws an ellipse with a minor radius of *SEMIMINORRADIUS* and a major radius of *SEMIMAJORRADIUS* about the point (*X*, *Y*) onto the destination bitmap of *DISPLAYSTREAM*. *ORIENTATION* is the angle of the major axis in degrees, positive in the counterclockwise direction. *DISPLAYSTREAM*'s position is left at (*X*, *Y*). (Dashing may not be implemented for this function yet.) The other arguments are interpreted as described above.

19.10 TYPESCRIPT FACILITIES: THE "T" FILE

Output to the file T and echoing of type-in is directed to a distinguished terminal display stream. This is initialized to be a display stream at the top of the screen, but that initial setting can be modified by the function TTYDISPLAYSTREAM.

(TTYDISPLAYSTREAM *DISPLAYSTREAM*). [Function]

Selects the display stream or window *DISPLAYSTREAM* to be the terminal output channel, and returns the previous terminal output display stream. TTYDISPLAYSTREAM puts *DISPLAYSTREAM* into scrolling mode and calls PAGEHEIGHT with the number of lines that will fit into *DISPLAYSTREAM* given its current Font and ClippingRegion. The linelength of TTYDISPLAYSTREAM is computed (like any other display stream) from its LeftMargin, RightMargin, and Font. If one of these fields is changed, its linelength is recalculated. If one of the fields used to compute the number of lines (such as the ClippingRegion or Font) changes, PAGEHEIGHT is not automatically recomputed. (TTYDISPLAYSTREAM (TTYDISPLAYSTREAM)) will cause it to be recomputed.

If the window system is active, the line buffer is saved in the old TTY window, and the line buffer is set to the one saved in the window of the new display stream, or to a newly created line buffer (if it does not have one). Caution: It is possible to move the TTYDISPLAYSTREAM to a nonvisible display stream or to a window whose current position is not in its clipping region.

(CARET *NEWCARET*) [Function]

Sets the shape that blinks at the location of the next output to the TTYDISPLAYSTREAM. *NEWCARET* is either (1) NIL - no changes, returns a CURSOR representing the current caret, (2) OFF - turns the caret off, or (3) a CURSOR which gives the new caret shape. The hotspot of *NEWCARET* indicates which point in the new caret bitmap should be located at the current output position. The previous caret is returned.

(PAGEHEIGHT *N*) [Function]

If *N* is greater than 0, it is the number of lines of output that will be printed to TTYDISPLAYSTREAM before the page is held. A page is held before the *N*+1 line is printed to TTYDISPLAYSTREAM without intervening input if there is no terminal input waiting to be read. The output is held with the screen video reversed until a character is typed. Output holding is disabled if *N* is 0. PAGEHEIGHT returns the previous setting.

19.11 CURSOR AND MOUSE

The screen relative position at which the cursor bitmap is being displayed can be read or set using the functions:

(CURSORPOSITION *NEWPOSITION DISPLAYSTREAM OLDPOSITION*) [Function]

This returns the location of the cursor in the coordinate system of *DISPLAYSTREAM* (the current display stream, if *DISPLAYSTREAM* is NIL). If *OLDPOSITION* is a POSITION, it will be reused, and returned. If *NEWPOSITION* is non-NIL, it should be a position and the cursor will be positioned at *NEWPOSITION*.

(ADJUSTCURSORPOSITION *DELTAX DELTAY*) [Function]

Moves the cursor *DELTAX* points in the X direction and *DELTAY* points in the Y direction. *DELTAX* and *DELTAY* default to 0.

The cursor can be changed like any other bitmap by BITBLTing into it or pointing a display stream at it and printing or drawing curves. For most applications, it is also necessary to locate the *hotspot* - a point within the CURSORWIDTH by CURSORHEIGHT area which is used to determine a *point* position for the cursor. Also for some applications it is necessary to save and restore the cursor. The Cursor record and the following functions provide these capabilities. A Cursor record has fields CURSORBITMAP and CURSORHOTSPOT, the latter a POSITION that gives the location of the hot spot inside the cursor.

(CURSORCREATE *BITMAP X Y*) [Function]

Returns a cursor object which has *BITMAP* as its image and the location (*X*,*Y*) as the hot spot. If *X* is a POSITION, it is used as the hot spot. If *BITMAP* has dimensions different from CURSORWIDTH by CURSORHEIGHT, the lesser of the widths and the lesser of the heights are used to determine the bits that actually get copied into the lower left corner of the cursor. If *X* is NIL, 0 is used. If *Y* is NIL, CURSORHEIGHT-1 is used. The default cursor is an uparrow with its tip in the upper left corner and its hot spot at (0,CURSORHEIGHT-1).

(CURSOR *NEWCURSOR* —) [Function]

Returns a CURSOR record instance that contains (a copy of) the current cursor specification. If *NEWCURSOR* is a CURSOR record instance, the cursor will be set to the values in *NEWCURSOR*. If *NEWCURSOR* is T, the cursor will be set to the default cursor DEFAULTCURSOR, an upward left pointing arrow.

(SETCURSOR *NEWCURSOR* —) [Function]

If *NEWCURSOR* is a CURSOR record instance, the cursor will be set to the values in *NEWCURSOR*. This does not return the old cursor, and therefore, provides a way of changing the cursor without using storage.

(FLIPCURSOR) [Function]

Inverts the cursor.

There are several cursors defined in Interlisp-D that may be of interest to users. One of these is WAITINGCURSOR, an hour glass shape used by the system to indicate that a long computation is in progress.

CURSORs can be saved on a file using the file package command CURSORS, or the UGLYVARS file package command.

19.11.1 Mouse Button Testing

There are various graphical input devices that can be read from Interlisp-D. The devices used in this manner are: a device called a mouse, which has three keys and steers the cursor, and seven uninterpreted keys on the keyboard. (Some Xerox 1100 systems may also have a small, five-key keyset.) The following macros are provided to test the state of these input devices. (The three keys on the mouse (often called buttons) are referred to by their location: left, middle, or right.)

(MOUSESTATE *BUTTONFORM*) [Macro]

Reads the mouse state and returns T if that state is described by *BUTTONFORM*.

BUTTONFORM can be one of the key indicators LEFT, MIDDLE, or RIGHT; the atom UP (indicating all keys are up); the form (ONLY *KEY*); or a form of AND, OR, or NOT applied to any valid button form. For example: (MOUSESTATE LEFT) will be true if the left mouse button is down. (MOUSESTATE (ONLY LEFT)) will be true if the left mouse button is the only one down. (MOUSESTATE (OR (NOT LEFT) MIDDLE)) will be true if either the left mouse button is up or the middle mouse button is down.

(LASTMOUSESTATE *BUTTONFORM*) [Macro]

Similar to MOUSESTATE, but tests the value of LASTMOUSEBUTTONS rather than getting the current state. This is useful for determining which keys caused a MOUSESTATE to be true.

(UNTILMOUSESTATE *BUTTONFORM INTERVAL*) [Macro]

BUTTONFORM is as described in MOUSESTATE. Waits until *BUTTONFORM* is true or until *INTERVAL* milliseconds have elapsed. The value of UNTILMOUSESTATE is T if *BUTTONFORM* was satisfied before it timed out, otherwise NIL. If *INTERVAL* is NIL, it waits indefinitely. It compiles into an open loop that calls the TTY wait background function. This form should not be used inside the TTY wait background function. UNTILMOUSESTATE does not use any storage during its wait loop.

The macros KEYSETSTATE and LASTKEYSETSTATE are identical to MOUSESTATE and LASTMOUSESTATE except that they also check the state of the five-finger keyset as well as the state of the mouse buttons. That is they check the state of both the mouse and the keyset. Thus, if the left mouse button was the only mouse button held down, (MOUSESTATE (ONLY LEFT)) would be T even though a keyset key was down; whereas (KEYSETSTATE (ONLY LEFT)) would be NIL if a keyset button were down.

The names of the keyset keys are: LEFTKEY, LEFTMIDDLEKEY, MIDDLEKEY, RIGHTMIDDLEKEY and RIGHTKEY.

19.11.2 Low Level Access to Mouse

This section describes the low level access to the graphical input devices and can be skipped by most users. Graphical input information is represented in the following global variables:

LASTMOUSEX [Variable]

The X position of the cursor in absolute screen coordinates. Also see the function LASTMOUSEX below.

LASTMOUSEY [Variable]

The Y position of the cursor in absolute screen coordinates. Also see the function LASTMOUSEY below.

LASTMOUSEBUTTONS [Variable]

An 8-bit number that has bits on corresponding to the mouse buttons that are down: 4Q is the left mouse button, 2Q is the right button, 1Q is the middle button. (Bits 200Q, 100Q, 40Q, 20Q, and 10Q give the state of the keyset keys, from left to right, if you have a keyset.)

LASTKEYBOARD [Variable]

The state of certain keys on the keyboard (200Q = lock, 100Q = left shift, 40Q = ctrl, 10Q = right shift, 4Q = blankBottom, 2Q = blankMiddle, 1Q = blankTop). If the key is down, the corresponding bit is on.

LASTMOUSETIME [Variable]

The time in milliseconds since the mouse was last read (since the last call to GETMOUSESTATE. LASTMOUSETIME is a 16-bit positive integer so it rolls over every 65+ seconds.

The following functions provide low level cursor access in display stream coordinates.

(LASTMOUSEX *DISPLAYSTREAM*) [Function]

Returns the value of the cursor's X position in the coordinates of *DISPLAYSTREAM*.

(LASTMOUSEY *DISPLAYSTREAM*) [Function]

Returns the value of the cursor's Y position in the coordinates of *DISPLAYSTREAM*.

(DECODEBUTTONS *BUTTONSTATE*) [Function]

Returns a list of the buttons or keys that are down in the state *BUTTONSTATE*. If *BUTTONSTATE* is not a SMALLP, LASTMOUSEBUTTONS is used (see GETMOUSESTATE below). The button names that can be returned are: LEFT, MIDDLE, RIGHT (the three mouse keys), LEFTKEY, LEFTMIDDLEKEY, MIDDLEKEY, RIGHTMIDDLEKEY and RIGHTKEY (the five keyset keys).

(GETMOUSESTATE) [Function]

Reads the current state of the mouse and sets the variables LASTMOUSEX, LASTMOUSEY, LASTMOUSEBUTTONS, LASTMOUSETIME, and LASTKEYBOARD. In polling mode, the program must remember the previous state and look for changes, such as a key going up or down, or the cursor moving outside a region of interest.

19.12 WINDOWS

Windows provide a means by which different programs can share the display harmoniously. Interlisp-D provides both interactive and programmatic constructs for creating, moving, reshaping, overlapping, and destroying windows in such a way that a program can be embedded in a window in a relatively transparent fashion. This is implemented by having each window save the bits that it obscures. This allows existing Interlisp programs to be used without change, while providing a base for experimentation with more complex window semantics in new applications.

Because the window system assumes that all programs follow certain conventions concerning control of the screen, ordinary user programs should not perform display operations directly on the screen. In particular, functions that can operate directly on bitmaps (such as BITBLT or BITMAPBIT) should not be given (SCREENBITMAP) as the destination argument. All interactions with the screen should take place through windows.

For specialized applications that require taking complete control of the display, the window system can be turned off (and back on again) with the following function:

(WINDOWWORLD *FLAG*) [NoSpread Function]

The window world is turned on if *FLAG* is T and off if *FLAG* is NIL. WINDOWWORLD returns the previous state of the window world (T or NIL). If WINDOWWORLD is given no arguments, it simply returns the current state without affecting the window world.

19.12.1 What are Windows?

A window specifies a region of the screen, a display stream, a location in an occlusion stack, functions that get called when the window undergoes certain actions, and various other items of information. The basic model is that a window is a passive collection of bits (on the screen). On top of this basic level, the system supports many different types of windows that are linked to the data structures displayed in them and provide selection and redisplaying routines. In addition, it is possible for the user to create new types of windows by providing selection and displaying functions for them.

Windows are ordered in depth from user to background. Windows in front of others obscure the latter. Operating on a window generally brings it to the top.

Windows are located at a certain position on the screen. Each window has a clipping region that confines all bits splashed at it to a region that allows a border around the window, and a title above it.

Each window has a display stream associated with it, and either a window or its display stream can be passed interchangeably to all system functions. There are dependencies between the window and its display stream that the user should not disturb. For instance, the destination bitmap of the display stream of a window must always be (SCREENBITMAP). The XOffset, YOffset, and ClippingRegion attributes of the display stream should not be changed. At some future date, the notions of window and display stream will be merged.

Windows can be created by the user interactively, under program control, or may be created automatically by the system.

Windows are in one of two states: "open" or "closed". In an "open" state, a window is on the occlusion stack and therefore visible on the screen (unless it is covered by other open windows) and accessible to mouse operations. In a "closed" state, a window is not on the occlusion stack and therefore not visible and not accessible to mouse operations. Any attempt to print or draw on a closed window will open it.

When Interlisp-D starts up, there are three windows on the screen: a top level typescript window, a window containing the Interlisp-D logo, and a prompt window. The top level typescript window corresponds to the file T in the EXEC process where the read-eval-print loop is operating. The logo window is bound to the variable LOGOW until it is closed. The prompt window is used for the printing of help or prompting messages. It is available to user programs through the following functions:

PROMPTWINDOW [Variable]

Global variable containing the prompt window.

(PROMPTPRINT *EXP*) [NoSpread Function]

Prints *EXP* in the prompt window.

(CLRPROMPT) [Function]

Clears the prompt window.

19.12.2 Interactive Window Operations

The Interlisp-D window system allows the user to interactively manipulate the windows on the screen, moving them around, changing their shape, etc. by selecting various operations from a menu. Programmatic versions of these operations are described on page 19.26.

For most windows, depressing the RIGHT mouse key when the cursor is inside a window during I/O wait will cause the window to come to the top and a menu of window operations to appear. If a command is selected from this menu (by releasing the right mouse key while the cursor is over a command), the selected operation will be applied to the window in which the menu was brought up. (It is possible for an applications program to redefine the action of the RIGHT mouse key. In these cases, there is a convention that the default command menu may be brought up by depressing the RIGHT key when the cursor is in the header or border of a window. See page 19.30) The operations are:

CLEAR [Window Menu Command]

Clears the window and repositions it to the left margin of the first line of text (below the upper left corner of the window by the amount of the font ascent).

CLOSE [Window Menu Command]

Closes the window, i.e, removes it from the screen. (See CLOSEW, page 19.26.)

BURY [Window Menu Command]

Puts the window on the bottom of the occlusion stack, thereby exposing any windows that it was hiding.

MOVE [Window Menu Command]

Moves the window to a location specified by depressing and then releasing the LEFT key. During this time a ghost frame will indicate where the window will reappear when the key is released. (See GETBOXPOSITION, page 19.36.)

SHAPE [Window Menu Command]

Allows the user to specify a new region for the existing window contents. If the LEFT key is used to specify the new region, the reshaped window can be placed anywhere. If the MIDDLE key is used, the cursor will start out tugging at the nearest corner of the existing window, which is useful for making small adjustments in a window that is already positioned correctly.

REDISPLAY [Window Menu Command]

Redisplays the window. (See REDISPLAYW, page 19.27.)

PAINT [Window Menu Command]

Switches to a mode in which the cursor can be used like a paint brush to draw in a window. This is useful for making notes on a window. While the LEFT key is down, bits are added. While the MIDDLE key is down, they are erased. The RIGHT button pops up a command menu that allows changing of the brush shape, size and shade, changing the mode of combining the brush with the existing bits, or stopping paint mode.

Paint mode also contains a hardcopy command that makes a Press file of the bits in a window and sends it to the printer. There are limitations on the complexity and size of the bitmaps that some printers will print. If the printer does not print the entire window correctly, try a smaller window or one with fewer black bits in it. To get a hardcopy of an arbitrary part of the screen that crosses window boundaries, use the HARDCOPY command in the background menu (below).

SNAP [Window Menu Command]

Prompts for a region on the screen and makes a new window whose bits are a snapshot of the bits currently in that region. Useful for saving some particularly choice image before the window image changes.

Occasionally, a user will have a number of large windows on the screen, making it difficult to access those windows being used. To help with the problem of screen space management, the Interlisp-D window system allows the creation of *Icons.* An icon is a small rectangle (containing text or a bitmap) which is a "shrunken-down" form of a particular window. Using the SHRINK and EXPAND commands, the user can shrink windows not currently being used into icons, and quickly restore the original windows at any time.

SHRINK [Window Menu Command]

Removes the window from the screen and brings up its icon. (See SHRINKW, page 19.27.) The window can be restored by selecting EXPAND from the window command menu of the icon.

If the RIGHT button is pressed while the cursor is in an icon, the window command menu will contain a slightly different set of commands. The REDISPLAY and CLEAR commands are removed, and the SHRINK command is replaced with the EXPAND command:

EXPAND [Window Menu Command]

Restores the window associated with this icon and removes the icon. (See EXPANDW, page 19.28.)

If the RIGHT button is pressed while the cursor is not in any window, a "background menu" appears with the following operations:

SAVEVM [Window Menu Command]

Calls the function SAVEVM (page 18.4), which writes out all of the dirty pages of the virtual memory. After a SAVEVM, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the SAVEVM) should you experience a system crash or other disaster.

SNAP [Window Menu Command]

The same as the SNAP command described above.

HARDCOPY [Window Menu Command]

Prompts for a region on the screen, makes a press file and sends it to the printer.

The printing is done with HARDCOPYW (page 18.18), so if FULLPRESSPRINTER is non-NIL, the image will be sent there, rather than to (PRINTINGHOST).

Some built-in facilities and Lispusers packages add commands to the background menu, to provide an easy way of calling the different facilities. The user can determine what these new commands do by holding the RIGHT button down for a few seconds over the item in question; an explanatory message will be printed in the prompt window.

The following functions provide a functional interface to the interactive window operations so that user programs can call them directly.

(DOWINDOWCOM *WINDOW*) [Function]

If *WINDOW* is NIL, it calls DOBACKGROUNDCOM. If *WINDOW* is a shrunken window, it brings up the "icon window" menu. If *WINDOW* is a unshrunken window, it brings up the window menu. The initial items in these menus are described above. If the user selects one of the items from the provided menu, that item is APPLYed to *WINDOW*. If *WINDOW* is not a WINDOW or NIL, it returns.

(DOBACKGROUNDCOM) [Function]

Brings up the background menu. The initial items in this menu are described above. If the user selects one of the items from the menu, that item is EVALed.

19.12.3 Changing Entries on the Window Command Menus

The window command menus for unshrunken windows, shrunken windows, and the background are cached in the variables WindowMenu, IconWindowMenu, and BackgroundMenu. To change the entries in these menus, the user should change the change the menu "command lists" in the variables WindowMenuCommands, IconWindowMenuCommands, and BackgroundMenuCommands, and set the appropriate menu variable to a non-MENU, so the menu will be recreated. This provides a way of adding commands to the menu, of changing its font or of restoring the menu if it gets clobbered. The "command lists" are in the format of the ITEMS field of a menu (see page 19.39), except as specified below.

Note: command menus are recreated using the current value of MENUFONT.

WindowMenu [Variable]
WindowMenuCommands [Variable]

The menu that is brought up in response to a right button in an unshrunken window is stored on the variable WindowMenu. If WindowMenu is set to a non-MENU, the menu will be recreated from the list of commands WindowMenuCommands. The CADR of each command added to WindowMenuCommands should be a function name that will be APPLYed to the window.

IconWindowMenu [Variable]
IconWindowMenuCommands [Variable]

The menu that is brought up in response to a right button in a shrunken window is stored on the variable IconWindowMenu. If it is NIL, it is recreated from the list of commands IconWindowMenuCommands. The CADR of each command added a function name that will be APPLYed to the window.

BackgroundMenu [Variable]
BackgroundMenuCommands [Variable]

The menu that is brought up in response to a right button in the background is stored on the variable BackgroundMenu. If it is NIL, it is recreated from the list of commands BackgroundMenuCommands. The CADR of each command added to BackgroundMenuCommands should be a form that will be EVALed.

19.12.4 Coordinate Systems

One way of thinking of a window is as a "view" onto an object (e.g. a graph, a file, a picture, etc.) The object has its own natural coordinate system in terms of which its subparts are laid out. When the window is created, the XOffset and YOffset of the window's display stream are set to map the origin of the object's coordinate system into the lower left point of the window's interior region. At the same time, the ClippingRegion of the display stream is set to correspond to the interior of the window. From then on, the display stream's coordinate system is translated and its clipping region adjusted whenever the window is moved, scrolled or reshaped.

There are several distinct regions associated with a window viewing an object. First, there is a region in the window's coordinate system that contains the complete image of the object. This region (which can only be determined by application programs with knowledge of the "semantics" of the object) is stored as the EXTENT property of the window (page 19.32). Second, the clipping region of the window (obtainable with the function DSPCLIPPINGREGION) specifies the portion of the object that is actually visible in the window. This is set so that it corresponds to the interior of the window (not including the border or title). Finally, there is the region on the screen that specifies the total area that the window occupies, including the border and title. This region (in screen coordinates) is stored as the REGION property of the window (page 19.33).

19.12.5 Scrolling

The window system supports the idea of scrolling the contents of a window. Scrolling regions are on the left and the bottom edge of each window. The scrolling regions will only be active if the window has a SCROLLFN window property (page 19.31). If a window has a SCROLLFN and the cursor moves from inside that window into its scrolling region and remains there for SCROLLWAITTIME milliseconds (initially 1000), a scroll bar appears. The value of the global variable SCROLLBARWIDTH (initially 24) determines the size of the scrolling region. The LEFT key is used to indicate upward or leftward scrolling by the amount necessary to move the selected position to the top or the left edge. The RIGHT key is used to indicate downward or rightward scrolling by the amount necessary to move the top or left edge to the selected position. The MIDDLE key is used to indicate global placement of the object within the window (similar to "thumbing" a book).

In the scroll region, the part of the object that is being viewed by the window is marked with a gray shade. If the whole scroll bar is thought of as the entire object, the shaded portion is the portion currently being viewed. This will only occur when the window "knows" how big the object is (see window property EXTENT, page 19.32).

When the button is released in a scroll region, the function SCROLLW is called. SCROLLW calls the scrolling function associated with the window to do the actual scrolling and provides a programmable entry to the scrolling operation.

(SCROLLW *WINDOW DELTAX DELTAY CONTINUOUSFLG*) [Function]

Calls the SCROLLFN window property of the window *WINDOW* with arguments *WINDOW*, *DELTAX*, *DELTAY* and *CONTINUOUSFLG*. See SCROLLFN window property, page 19.31.

The function that tracks the mouse while it is in the scroll region is SCROLL.HANDLER.

(SCROLL.HANDLER *WINDOW*) [Function]

This is called when the cursor leaves a window in either the left or downward direction. If *WINDOW* does not have a scroll region for this direction (e.g. the window has moved or reshaped since it was last scrolled), a scroll region is created that is SCROLLBARWIDTH wide. It then waits for SCROLLWAITTIME milliseconds and if the cursor is still inside the scroll region, it opens a window the size of the scroll region and changes the cursor to indicate the scrolling is taking place.

When a button is pressed, the cursor shape is changed to indicate the type of scrolling (up, down, left, right or thumb). After the button is held for WAITBEFORESCROLLTIME milliseconds, until the button is released SCROLLW is called each WAITBETWEENSCROLLTIME milliseconds. These calls are made with the *CONTINUOUSFLG* argument set to T. If the button is released before WAITBEFORESCROLLTIME milliseconds, SCROLLW is called with the *CONTINUOUSFLG* argument set to NIL.

The arguments passed to SCROLLW depend on the mouse button. If the LEFT button is used in the vertical scroll region, *DY* is distance from cursor position at the time the button was released to the top of the window and *DX* is 0. If the RIGHT button is used, the inverse of this quantity is used for *DY* and 0 for *DX*. If the LEFT button is used in the horizontal scroll region, *DX* is distance from cursor position to left of the window and *DY* is 0. If the RIGHT button is used, the inverse of this quantity is used for *DX* and 0 for *DY*.

If the MIDDLE button is pressed, the distance argument to SCROLLW will be a FLOATP between 0.0 and 1.0 that indicates the proportion of the distance the cursor was from the left or top edge to the right or bottom edge.

SCROLLBYREPAINTFN is the standard scrolling function which should be used as the SCROLLFN property for most scrolling windows.

(SCROLLBYREPAINTFN *WINDOW DELTAX DELTAY CONTINUOUSFLG*) [Function]

This function, when used as a SCROLLFN, BITBLTs the bits that will remain visible after the scroll to their new location, fills the newly exposed area with texture, adjusts the window's coordinates and then calls the window's REPAINTFN on the newly exposed region. Thus this function will scroll any window that has a repaint function. If *WINDOW* has an EXTENT property (page 19.32), SCROLLBYREPAINTFN will limit scrolling to keep the extent region visible or near visible. That is, it will not scroll the window so that the top of the extent is below the top of the window, the bottom of the extent is more than one point above the top of the window, the left of the extent is to the right of the window and the right of the extent is to the left of the window. The EXTENT is scrolled to just above the window to provide a way of "hiding" the contents of a window.

If *DELTAX* or *DELTAY* is a FLOATP, SCROLLBYREPAINTFN will position the window so that its top or left edge will be positioned at that proportion of its EXTENT. If the window does not have an EXTENT, SCROLLBYREPAINTFN will do nothing.

If *CONTINUOUSFLG* is non-NIL, this indicates that the scrolling button is being held down. In this case, SCROLLBYREPAINTFN will scroll the distance of one linefeed height (as returned by DSPLINEFEED, page 19.12).

19.12.6 Programmatic Window Operations

(CREATEW *REGION TITLE BORDER NOOPENFLG*) [Function]

Creates a new window. *REGION* indicates where and how large the window should be by specifying the exterior region of the window (the usable height and width of the resulting window will be smaller than the height and width of the region by twice the border size and further less the height of the title, if any). If *REGION* is NIL, GETREGION is called to prompt the user for a region.

If *TITLE* is non-NIL, it is printed in the border at the top of the window. The *TITLE* is printed using the global display stream WindowTitleDisplayStream. Thus the height of the title will be (FONTPROP WindowTitleDisplayStream 'HEIGHT).

If *BORDER* is a number, it is used as the border size. If *BORDER* is not a number, the window will have a border WBorder (initially 4) bits wide.

If *NOOPENFLG* is non-NIL, the window will not be opened, i.e. displayed on the screen.

(WINDOWP *X*) [Function]

Returns *X* if *X* is a window, NIL otherwise.

(OPENWP *WINDOW*) [Function]

Returns *WINDOW*, if *WINDOW* is an open window (has not been closed); NIL otherwise.

(OPENWINDOWS) [Function]

Returns a list of all active windows.

(WHICHW *X Y*) [Function]

Returns the window which contains the position in screen coordinates of *X* if *X* is a POSITION, the position (*X*,*Y*) if *X* and *Y* are numbers, or the position of the cursor if *X* is NIL. Returns NIL if the coordinates are not in any window. If they are in more than one window, it returns the uppermost.

Example: (WHICHW) returns the window that the cursor is in.

(DECODE/WINDOW/OR/DISPLAYSTREAM *DSORW WINDOWVAR TITLE BORDER*) [Function]

If *DSORW* is a display stream, it is returned. If *DSORW* is a window, its display stream is returned. If *DSORW* is NIL, it evaluates *WINDOWVAR* (which should be an atom). If its value is a window, it is reopened if it is closed, and returned. If its value is not a window, *WINDOWVAR* is set to a newly created window (prompting user for region) and returned. If *DSORW* is NEW, a new window is created and returned. If *TITLE* or *BORDER* are given and a window is involved, the TITLE or BORDER property of the window is reset. The *DSORW*=NIL case is most useful for programs that want to display their output in a window, but want to reuse the same window each time they are called. The non-NIL cases are good for decoding a display stream argument passed to a function.

(WIDTHIFWINDOW *INTERIORWIDTH BORDER*) [Function]

Returns the width of the window necessary to have *INTERIORWIDTH* points in its interior if the width of the border is *BORDER*. If *BORDER* is NIL, the default border size WBorder is used.

(HEIGHTIFWINDOW *INTERIORHEIGHT TITLEFLG BORDER*) [Function]

Returns the height of the window necessary to have *INTERIORHEIGHT* points in its interior with a border of *BORDER* and, if *TITLEFLG* is non-NIL, a title. If *BORDER* is NIL, the default border size WBorder is used.

WIDTHIFWINDOW and HEIGHTIFWINDOW are useful for calculating the width and height for a call to GETBOXPOSITION for the purpose of positioning a window.

Interlisp-D provides a set of operations which apply to any window. In addition to being available as functions, most of these are also available via the standard mouse interface. See page 19.20

(TOTOPW *WINDOW NOCALLTOPWFN*) [Function]

Brings *WINDOW* to the top of the stack of overlapping windows, guaranteeing that it is entirely visible. If *WINDOW* is closed, it is opened. This is done automatically whenever a printing or drawing operation occurs to the window.

If *NOCALLTOPWFN* is NIL, the TOTOPFN of *WINDOW* is called (page 19.30). If *NOCALLTOPWFN* is T, it is not called, which allows a TOTOPFN to call TOTOPW without causing an infinite loop.

(SHAPEW *WINDOW NEWREGION*) [Function]

Reshapes *WINDOW* to the region *NEWREGION*, or prompts for a region (with GETREGION, page 19.37) if none is supplied. Calls the window's RESHAPEFN, if any (page 19.31).

(CLOSEW *WINDOW*) [Function]

CLOSEW calls the function or functions on the window property CLOSEFN of *WINDOW*, if any (page 19.30). If one of the CLOSEFNs is the atom DON'T or returns the atom DON'T as a value, CLOSEW returns without doing anything further. Otherwise, CLOSEW removes *WINDOW* from the window stack and restores the bits it is obscuring. If *WINDOW* was closed, *WINDOW* is returned as the value. If it was not closed, (for example because its CLOSEFN returned the atom DON'T), NIL is returned as the value.

WINDOW can be restored in the same place with the same contents (reopened) by calling OPENW or by using it as the source of a display operation.

(OPENW *WINDOW*) [Function]

If *WINDOW* is a closed window, OPENW calls the function or functions on the window property OPENFN of *WINDOW*, if any (page 19.30). If one of the OPENFNs is the atom DON'T, the window will not be opened. Otherwise the window is placed on the occlusion stack of windows and its contents displayed on the screen. If *WINDOW* is an open window, it returns NIL.

(MOVEW *WINDOW POSorX Y*) [Function]

Moves *WINDOW* to the position specified by *POSorX* and *Y* according to the following rules:

If *POSorX* is NIL, GETBOXPOSITION (page 19.36) is called to read a position from the user.

If *POSorX* is a POSITION, *POSorX* is used.

If *POSorX* and *Y* are both NUMBERP, a position is created using *POSorX* as the XCOORD and *Y* as the YCOORD.

If *POSorX* is a REGION, a position is created using its LEFT as the XCOORD and BOTTOM as the YCOORD.

If *WINDOW* is not open and *POSorX* is non-NIL, the window will be moved without being opened. Otherwise, it will be opened.

If *WINDOW* has the atom DON'T as a MOVEFN property (page 19.32), the window will not be moved. If *WINDOW* has any other non-NIL value as a MOVEFN property, it should be a function or list of functions that will be called before the window is moved with the *WINDOW* as an argument. If it returns the atom DON'T, the window will not be moved. If it returns a position, the window will be moved to that position instead of the one specified by *POSorX* and *Y*. If there are more than one MOVEFNs, the last one to return a value is the one that determines where the window is moved to.

If *WINDOW* is moved and *WINDOW* has a window property of AFTERMOVEFN (page 19.32), it should be a function or a list of functions that will be called after the window is moved with *WINDOW* as an argument.

MOVEW returns the new position, or NIL if the window could not be moved.

(RELMOVEW *WINDOW POSITION*) [Function]

Like MOVEW for moving windows but *POSITION* is interpreted relative to the current position of *WINDOW*. Example: The following code moves *WINDOW* to the right one screen point.

```
(RELMOVEW window (create POSITION XCOORD ← 1 YCOORD ← 0))
```

(CLEARW *WINDOW*) [Function]

Fills *WINDOW* with its background texture, changes its coordinate system so that the origin is the lower left corner of the window, sets its X position to the left margin and sets its Y position to the base line of the uppermost line of text, ie. the top of the window less the font ascent.

(BURYW *WINDOW*) [Function]

Puts *WINDOW* on the bottom of the stack by moving all the windows that it covers in front of it.

(REDISPLAYW *WINDOW REGION ALWAYSFLG*) [Function]

Redisplay the region *REGION* of the window *WINDOW*. If *REGION* is NIL, the entire window is redisplayed. If *ALWAYSFLG* is NIL, and *WINDOW* doesn't have a REPAINTFN (page 19.32), *WINDOW* will not change and the message "That window doesn't have a REPAINTFN" will be printed in the prompt window.

(SHRINKW *WINDOW TOWHAT ICONPOSITION EXPANDFN*) [Function]

SHRINKW makes a small icon which represents *WINDOW* and removes *WINDOW* from the screen. Icons have a different window command menu that contains "EXPAND" instead of "SHRINK". The EXPAND command calls EXPANDW which returns the shrunken window to its original size and place.

The SHRINKFN property (page 19.30) of the window *WINDOW* affects the operation of SHRINKW. If the SHRINKFN property of *WINDOW* is the atom DON'T, SHRINKW prints "Can't shrink that window" in the PROMPTWINDOW and returns. Otherwise, the SHRINKFN property of the window is treated as a (list of) function(s) to apply to *WINDOW*; if any returns the atom DON'T, SHRINKW prints "Can't shrink that window" in the PROMPTWINDOW and returns.

TOWHAT, if given, indicates the image the icon window will have. If *TOWHAT* is a string, atom or list, the icon's image will be that string (currently implemented as a title-only window with *TOWHAT* as the title.) If *TOWHAT* is a BITMAP, the icon's image will be a copy of the bitmap. If *TOWHAT* is a WINDOW, that window will be used as the icon.

If *TOWHAT* is not given (as is the case when invoked from the SHRINK window command), then the following apply in turn: (1) If the window has an ICONFN property (page 19.31), it gets called with arguments (*WINDOW* *OLDICON*), where *WINDOW* is the window being shrunk and *OLDICON* is the previously created icon, if any. The ICONFN should return one of the *TOWHAT* entities described above or return the *OLDICON* if it does not want to change it. (2) If the window has an ICON property (page 19.31), it is used as the value of *TOWHAT*. (3) If the window has neither an ICONFN or ICON property, the icon will be *WINDOW*'s title or, if *WINDOW* doesn't have a title, the date and time of the icon creation.

ICONPOSITION gives the position that the new icon will be on the screen. If it is NIL, the icon will be in the corner of the window furthest from the center of the screen.

In all cases the icon is cached on the property ICONWINDOW (page 19.31) of *WINDOW* so repeating SHRINKW reuses the same icon (unless overridden by the ICONFN described above). Thus to change the icon it is necessary to remove the ICONWINDOW property or call SHRINKW explicitly giving a *TOWHAT* argument.

(EXPANDW *ICON*) [Function]

Restores the window for which *ICON* is an icon, and removes the icon from the screen. If the EXPANDFN (page 19.31) window property of the main window is the atom DON'T, the window won't be expanded. Otherwise, the window will be restored to its original size and location and the EXPANDFN (or list of functions) will be applied to it.

19.12.7 Window Properties

The behavior of a window is controlled by a set of window properties. Some of these are used by the system. However, any arbitrary property name may be used by a user program to associate information with a window. For many applications the user will associate the structure being displayed with its window using a property. The following functions provide for reading and setting window properties:

(WINDOWPROP *WINDOW PROP NEWVALUE*) [NoSpread Function]

Returns the previous value of *WINDOW*'s *PROP* aspect. If *NEWVALUE* is given, (even if given as NIL), it is stored as the new *PROP* aspect. Some aspects cannot be set by the user and will generate errors. Any *PROP* name that is not recognized is stored on a property list associated with the window.

(WINDOWADDPROP *WINDOW PROP ITEMTOADD*) [Function]

WINDOWADDPROP adds a new item to a window property. If *ITEMTOADD* is EQ to an element of the *PROP* property of the window *WINDOW*, nothing is added. If the current property is not a list, it is made a list before *ITEMTOADD* added. WINDOWADDPROP returns the previous property. The new item always goes on the end of the list. (Note: If the order of items in the list is important, the list can be modified using WINDOWPROP.) WINDOWADDPROP is useful for adding OPENFN or CLOSEFN functions to a window without affecting its existing functions.

(WINDOWDELPROP *WINDOW PROP ITEMTODELETE*) [Function]

WINDOWDELPROP deletes *ITEMTODELETE* from the window property *PROP* of *WINDOW* and returns the previous list if *ITEMTODELETE* was an element. If *ITEMTODELETE* was not a member of window property *PROP*, NIL is returned.

19.12.7.1 Mouse Function Window Properties

These properties allow the user to control the response to mouse activity in a window. The value of these properties, if non-NIL, should be a function that will be called (with the window as argument) when the specified event occurs.

Note: these functions should be "self-contained", communicating with the outside world solely via their window argument, e.g., by setting window properties. In particular, these functions should not expect to access variables bound on the stack, as the stack context is formally undefined at the time these functions are called. Since the functions are invoked asynchronously, they perform any TTY input operations from their own window.

WINDOWENTRYFN [Window Property]

Whenever a button goes down in the window and the process associated with the window (stored under the PROCESS property) is not the tty process, the WINDOWENTRYFN is called. The default is GIVE.TTY.PROCESS (page 18.34) which gives the process associated with the window the tty and calls the BUTTONEVENTFN.

CURSORINFN [Window Property]

Whenever the mouse moves into the window, the CURSORINFN is called.

CURSOROUTFN [Window Property]

The CURSOROUTFN is called when the cursor leaves the window.

CURSORMOVEDFN [Window Property]

The CURSORMOVEDFN is called whenever the cursor has moved and is inside the window. This allows a window function to implement "active" regions within itself by having its CURSORMOVEDFN determine if the cursor is in a region of interest, and if so, perform some action.

BUTTONEVENTFN [Window Property]

The BUTTONEVENTFN is called whenever there is a change in the state (up or down) of the mouse buttons inside the window. Changes to the mouse state while the BUTTONEVENTFN is running will not be interpreted as new button events, and the BUTTONEVENTFN will not be re-invoked.

RIGHTBUTTONFN [Window Property]

The RIGHTBUTTONFN is called in lieu of the standard window menu operation (DOWINDOWCOM) when the RIGHT key is depressed in a window. More specifically, the RIGHTBUTTONFN is called instead of the BUTTONEVENTFN when (MOUSESTATE (ONLY RIGHT)). If the RIGHT key is to be treated like any other key in a window, supply RIGHTBUTTONFN and BUTTONEVENTFN with the same function.

Note: When an application program defines its own RIGHTBUTTONFN, there is a convention that the default RIGHTBUTTONFN, DOWINDOWCOM (page 19.22), may be executed by depressing the RIGHT key when the cursor is in the header or border of a window. User programs are encouraged to follow this convention.

19.12.7.2 Event Window Properties

CLOSEFN [Window Property]

The CLOSEFN window property can be a single function or a list of functions that are called just before a window is closed by CLOSEW (page 19.26). (Note: If the CAR of the list is a LAMBDA word, it is treated as a single function.) The function(s) will be called with the window as a single argument. If any of the CLOSEFNs are the atom DON'T, or if the value returned by any of the CLOSEFNs is the atom DON'T, the window will not be closed.

Note: A CLOSEFN should not call CLOSEW on its argument.

OPENFN [Window Property]

The OPENFN window property can be a single function or a list of functions. If one of the OPENFNs is the atom DON'T, the window will not be opened. Otherwise, the OPENFNs are called after a window has been opened by OPENW (page 19.26), with the window as a single argument.

TOTOPFN [Window Property]

If non-NIL, whenever the window is brought to the top, the TOTOPFN is called (with the window as a single argument). This function may be used to bring a collection of windows to the top together.

If the *NOCALLTOPWFN* argument of TOTOPW (page 19.26) is non-NIL, the TOTOPFN of the window is not called, which provides a way of avoiding infinite loops when using TOTOPW from within a TOTOPFN.

SHRINKFN [Window Property]

The SHRINKFN window property can be a single function or a list of functions that are called just before a window is shrunken by SHRINKW (page 19.27), with the window as a single argument. If any of the SHRINKFNs are the atom DON'T, or if the value returned by any of the CLOSEFNs is the atom DON'T, the window will not be shrunk.

ICONFN [Window Property]

If SHRINKW (page 19.27) is called without begin given a *TOWHAT* argument (as is the case when invoked from the SHRINK window command) and the window's ICONFN property is non-NIL, then it gets called with two arguments, the window being shrunk and the previously created icon, if any. The ICONFN should return one of the *TOWHAT* entities described on page 19.27 or return the previously created icon if it does not want to change it.

ICON [Window Property]

If SHRINKW (page 19.27) is called without being given a *TOWHAT* argument, the window's ICONFN property is NIL, and the ICON property is non-NIL, then it is used as the value of *TOWHAT*.

ICONWINDOW [Window Property]

Whenever an icon is created, it is cached on the property ICONWINDOW of the window, so calling SHRINKW again will reuse the same icon (unless overridden by the ICONFN.

Thus, to change the icon it is necessary to remove the ICONWINDOW property or call SHRINKW (page 19.27) explicitly giving a *TOWHAT* argument.

EXPANDFN [Window Property]

The EXPANDFN window property can be a single function or a list of functions. If one of the EXPANDFNs is the atom DON'T, the window will not be expanded. Otherwise, the EXPANDFNs are called after the window has been expanded by EXPANDW (page 19.28), with the window as a single argument.

SCROLLFN [Window Property]

If the SCROLLFN property is NIL, the window will not scroll. Otherwise, it should be a function of four arguments: (1) the window being scrolled, (2) the distance to scroll in the horizontal direction (positive to right, negative to left), (3) the distance to scroll in the vertical direction (positive up, negative down), and (4) a flag which is T if the scrolling button is being held down. For more information, see SCROLL.HANDLER (page 19.24). For most scrolling windows, the SCROLLFN function should be SCROLLBYREPAINTFN (page 19.24).

NEWREGIONFN [Window Property]

The NEWREGIONFN is passed as the *NEWREGIONFN* argument to GETREGION (page 19.37) when the window is reshaped.

RESHAPEFN [Window Property]

The RESHAPEFN window property can be a single function or a list of functions that are called when a window is reshaped by SHAPEW (page 19.26). If the RESHAPEFN is DON'T or a list containing DON'T, the window will not be reshaped. Otherwise, the function(s) are called after the window has been reshaped, its coordinate system readjusted to the new position, the title and border displayed, and the interior filled with texture. The RESHAPEFN should display any additional information needed to complete the window's image in the new position and shape. The RESHAPEFN is called with three arguments: (1) the window in its reshaped form, (2) a bitmap with the contents of the old window, and (3) the region within the bitmap that contains the old image. This function is provided so that users can reformat window contents or whatever. RESHAPEBYREPAINTFN (page 19.33) is the default and should be useful for many windows.

REPAINTFN [Window Property]

The REPAINTFN window property can be a single function or a list of functions that are called to repaint parts of the window by REDISPLAYW (page 19.27). The REPAINTFNs are called with two arguments: the window and the region in the coordinates of the window's display stream of the area that should be repainted. Before the REPAINTFN is called, the clipping region of the window is set to clip all display operations to the area of interest so that the REPAINTFN can display the entire window contents and the results will be appropriately clipped. (Note: CLEARW (page 19.27) should not be used in REPAINTFNs because it resets the window's coordinate system. If a REPAINTFN wants to clear its region first, it should use DSPFILL (page 19.12).)

MOVEFN [Window Property]

If the MOVEFN is DON'T, the window will not be moved by MOVEW (page 19.26). Otherwise, if the MOVEFN is non-NIL, it should be a function or a list of functions that will be called before a window is moved with two arguments: the window being moved and the new position of the lower left corner in screen coordinates. If the MOVEFN returns DON'T, the window will not be moved. If the MOVEFN returns a POSITION, the window will be moved to that position. Otherwise, the window will be moved to the specified new position.

AFTERMOVEFN [Window Property]

If non-NIL, it should be a function or a list of functions that will be called after the window is moved (by MOVEW, page 19.26) with the window as an argument.

19.12.7.3 Miscellaneous Properties

TITLE [Window Property]

Accesses the title of the window. If a title is added to a window whose title is NIL or the title is removed (set to NIL) from a window with a title, the window's exterior (its region on the screen) is enlarged or reduced to accomodate the change without changing the window's interior. For example, (WINDOWPROP *WINDOW* 'TITLE "Results") changes the title of *WINDOW* to be "Results". (WINDOWPROP *WINDOW* 'TITLE NIL) removes the title of *WINDOW*.

BORDER [Window Property]

Accesses the width of the border of the window. The border will have at most 2 point of white (but never more than half) and the rest black. The default border is the value of the global variable WBorder (initially 4).

EXTENT [Window Property]

Used to limit scrolling operations (see page 19.23). Accesses the extent region of the window. If non-NIL, the EXTENT is a region in the window's display stream that contains the complete image of the object being viewed by the window. User programs are responsible for updating the EXTENT. The functions UNIONREGIONS, EXTENDREGION, etc. (page 19.3) are useful for computing a new extent region.

In some situations, it is useful to define an EXTENT that only exists in one dimension. This may be done by specifying an EXTENT region with a width or height of -1. SCROLLFN handling recognizes this situation as meaning that the negative EXTENT dimension is unknown.

PROCESS [Window Property]

If the PROCESS window property is non-NIL, it should be a PROCESS and will be made the TTY process by GIVE.TTY.PROCESS (page 18.34), the default WINDOWENTRYFN property. This implements the mechanism by which the keyboard is associated with different processes.

PAGEFULLFN [Window Property]

If the PAGEFULLFN is non-NIL, it will be called with the window as a single argument when the window is full (i.e., when enough has been printed since the last TTY interaction so that the next character printed will cause information to be scrolled off the top of the window.) If the PAGEFULLFN is NIL, the system function PAGEFULLFN (page 19.33) is called.

Note: PAGEFULLFN is only called on windows which are the TTYDISPLAYSTREAM of some process (see page 19.15).

The following properties are read-only (i.e. their property values cannot be changed using WINDOWPROP.

DSP [Window Property]

Value is the display stream of the window. All system functions will operate on either the window or its display stream.

HEIGHT [Window Property]
WIDTH [Window Property]

Value is the height and width of the interior of the window (the usable space not counting the border and title).

REGION [Window Property]

Value is a region (in screen coordinates) indicating where the window (counting the border and title) is located on the screen.

19.12.8 Auxiliary Functions

(RESHAPEBYREPAINTFN *WINDOW OLDIMAGE OLDREGION*) [Function]

It BITBLTs the old region contents into the lower left corner of the new region. If the new shape is larger in either or both dimensions, the new areas exposed are to the top and right of the old image. When this happens, RESHAPEBYREPAINTFN calls *WINDOW*'s REPAINTFN (page 19.32) to display the newly exposed region's contents. Note that this may result in two calls to the REPAINTFN.

(PAGEFULLFN *WINDOW*) [Function]

If the window property PAGEFULLFN (page 19.33) is NIL, when the window is full the system function PAGEFULLFN is called. PAGEFULLFN simply returns if there are characters in the type-in buffer for *WINDOW*, otherwise it inverts the window and waits for the user to type a character. PAGEFULLFN is user advisable.

19.12.9 Example: A Scrollable Window

The following is a simple example showing how one might create a scrollable window.

CREATE.PPWINDOW creates a window that displays the pretty printed expression EXPR. The window properties PPEXPR, PPORIGX, and PPORIGY are used for saving this expression, and the initial window position. Using this information, REPAINT.PPWINDOW simply reinitializes the window position, and prettyprints the expression again. Note that the whole expression is reformatted every time, even if only a small part actually lies within the window. If this window was going to be used to display very large structures, it would be desirable to implement a more sophisticated REPAINTFN that only redisplays that part of the expression within the window. However, this scheme would be satisfactory if most of the items to be displayed are small.

RESHAPE.PPWINDOW resets the window (and stores the initial window position), calls REPAINT.PPWINDOW to display the window's expression, and then sets the EXTENT property of the window so that SCROLLBYREPAINTFN will be able to handle scrolling and "thumbing" correctly.

```
(DEFINEQ

(CREATE.PPWINDOW
  [LAMBDA (EXPR)                   (* rrb “ 4-OCT-82 12:06”)
                                   (* creates a window that displays
                                   a pretty printed expression.)

    (PROG (WINDOW)                 (* ask the user for a piece of the
                                   screen and make it into a window.)
          (SETQ WINDOW (CREATEW NIL "PP window"))
                                   (* put the expression on the
                                   property list of the window so that
                                   the repaint and reshape functions
                                   can access it.)
          (WINDOWPROP WINDOW (QUOTE PPEXPR)
                      EXPR)        (* set the repaint and reshape
                                   functions.)
          (WINDOWPROP WINDOW (QUOTE REPAINTFN)
                      (FUNCTION REPAINT.PPWINDOW))
          (WINDOWPROP WINDOW (QUOTE RESHAPEFN)
                      (FUNCTION RESHAPE.PPWINDOW))
                                   (* make the scroll function
                                   SCROLLBYREPAINTFN, a system
                                   function that uses the repaint
                                   function to do scrolling.)
          (WINDOWPROP WINDOW (QUOTE SCROLLFN)
                      (FUNCTION SCROLLBYREPAINTFN))
                                   (* call the reshape function to
                                   initially print the expression and
                                   calculate its extent.)
          (RESHAPE.PPWINDOW WINDOW)
```

```
     (RETURN WINDOW])


(REPAINT.PPWINDOW
 [LAMBDA (WINDOW REGION)            (* rrb " 4-OCT-82 11:52")

     (* the repainting function for a window with a pretty printed expression.
     This repainting function ignores the region to be repainted and repaints
     the entire window.)

                                    (* set the window position to the
                                    beginning of the pretty printing
                                    of the expression.)
   (MOVETO (WINDOWPROP WINDOW (QUOTE PPORIGX))
           (WINDOWPROP WINDOW (QUOTE PPORIGY))
           WINDOW)
   (PRINTDEF (WINDOWPROP WINDOW (QUOTE PPEXPR))
             0 NIL NIL NIL WINDOW])


(RESHAPE.PPWINDOW
 [LAMBDA (WINDOW)                   (* rrb " 4-OCT-82 12:01")
                                    (* the reshape function for a
                                    window with a pretty printed
                                    expression.)
   (PROG (BTM)

     (* set the position of the window so that the first character appears in
     the upper left corner and save the X and Y for the repaint function.)

     (DSPRESET WINDOW)
     (WINDOWPROP WINDOW (QUOTE PPORIGX)
                 (DSPXPOSITION NIL WINDOW))
     (WINDOWPROP WINDOW (QUOTE PPORIGY)
                 (DSPYPOSITION NIL WINDOW))
                                    (* call the repaint function to
                                    pretty print the expression in
                                    the newly cleared window.)
     (REPAINT.PPWINDOW WINDOW)

     (* save the region actually covered by the pretty printed expression so
     that the scrolling routines will know where to stop. The pretty printing
     of the expression does a carriage return after the last piece of the
     expression printed so that the current position is the base line of
     the next line of text. Hence the last visible piece of the expression
     (BTM) is the ending position plus the height of the font above the
     base line e.g its ASCENT.)

     (WINDOWPROP WINDOW (QUOTE EXTENT)
       (create REGION
           LEFT ← 0
```

```
                    BOTTOM ←[SETQ BTM (IPLUS (DSPYPOSITION NIL WINDOW)
                                             (FONTPROP WINDOW (QUOTE ASCENT]

                    WIDTH ←(WINDOWPROP WINDOW (QUOTE WIDTH))
                    HEIGHT ←(IDIFFERENCE (WINDOWPROP WINDOW (QUOTE HEIGHT))

                                           BTM])
)
```

19.13 INTERACTIVE DISPLAY FUNCTIONS

The following functions allow the user to interactively specify positions or regions on the display screen.

(GETPOSITION *WINDOW CURSOR*) [Function]

Returns a POSITION that is specified by the user. GETPOSITION waits for the user to press and release the left button of the mouse and returns the cursor position at the time of release. If *WINDOW* is a WINDOW, the position will be in the coordinate system of *WINDOW*'s display stream. If *WINDOW* is NIL, the position will be in screen coordinates. If *CURSOR* is a CURSOR, the cursor will be changed to it while GETPOSITION is running. If *CURSOR* is NIL, the value of the system variable CROSSHAIRS will be used as the cursor.

(GETBOXPOSITION *WIDTH HEIGHT ORGX ORGY WINDOW PROMPTMSG*) [Function]

Allows the user to position a "ghost" region of size *WIDTH* by *HEIGHT* on the screen, and returns the POSITION of the lower left corner of the region. If *PROMPTMSG* is non-NIL, GETBOXPOSITION first prints it in the PROMPTWINDOW. GETBOXPOSITION then changes the cursor to a box (using the global variable BOXCURSOR). If *ORGX* and *ORGY* are numbers, they are taken to be the original position of the region, and the cursor is moved to the nearest corner of that region. The user is then free to move the cursor around the screen. When a mouse button is depressed, a ghost region is locked to the cursor so that if the cursor is moved, the ghost region moves with it. If *ORGX* and *ORGY* are numbers, the corner of the original region that is nearest the cursor position at the time the button is pressed is locked, otherwise the lower left corner is locked. The user can change to another corner by continuing to hold down the left button and holding down the right button also. With both buttons down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the mouse will snap to the nearest corner, which will then become locked to the cursor. When all buttons are released, the lower left corner of the region is returned. If *WINDOW* is a WINDOW, the returned position will be in *WINDOW*'s coordinate system; otherwise it will be in screen coordinates.

Example:

```
(GETBOXPOSITION 100 200 NIL NIL NIL
     "Specify the position of the command area.")
```

prompts the user for a 100 wide by 200 high region and returns its lower left corner in screen coordinates.

(GETREGION *MINWIDTH MINHEIGHT INITREGION NEWREGIONFN NEWREGIONFNARG*) [Function]

Lets the user specify a new region and returns that region in screen coordinates. GETREGION prompts for a region by displaying a four-pronged box next to the cursor arrow. If the user presses the left button, one corner of a "ghost" region outline is locked to that point and the opposite corner is locked to the cursor. As the cursor moves, the outline expands. To specify a region, the user moves the cursor to one corner of the intended region, presses the left button, moves the cursor to the opposite corner while holding down the left button, and then releases the button.

If *INITREGION* is a REGION and the user presses the middle button, the corner of *INITREGION* farthest from the cursor position is fixed and the corner nearest the cursor is locked to the cursor.

One can switch from one corner to another while positioning the region. To change to another corner, continue to hold down the left button and hold down the right button also. With both buttons down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the cursor will snap to the nearest corner, which will become the moving corner. In this way, the region may be moved all over the screen, before its size and position is finalized.

MINWIDTH and *MINHEIGHT*, if given, are the smallest WIDTH and HEIGHT that the returned region will have. If the user specified region is smaller, it will be increased in width or height to these dimensions.

If *NEWREGIONFN* is non-NIL, it will be called to determine values for the positions of the corners. This provides a way of "filtering" prospective regions; for instance, by restricting the region to lie on an arbitrary grid. When the user is specifying a region, the region is determined by two of its corners, one that is fixed and one that is tracking the cursor. Each time the cursor moves or a mouse button is pressed, *NEWREGIONFN* is called with three arguments: *FIXEDPOINT*, the position of the fixed corner of the prospective region; *MOVINGPOINT*, the position of the opposite corner of the prospective region; and *NEWREGIONFNARG*. *NEWREGIONFNARG* allows the caller of GETREGION to pass information to the *NEWREGIONFN*. The first time a button is pressed, *MOVINGPOINT* is NIL and *FIXEDPOINT* is the position the user selected for the fixed corner of the new region. In this case, the position returned by *NEWREGIONFN* will be used for the fixed corner instead of the one proposed by the user. For all other calls, *FIXEDPOINT* is the position of the fixed corner (as returned by the previous call) and *MOVINGPOINT* is the new position the user selected for the opposite corner. In these cases, the value of *NEWREGIONFN* is used for the opposite corner instead of the one proposed by the user. In all cases, the ghost region is drawn with the values returned by *NEWREGIONFN*.

(GETBOXREGION *WIDTH HEIGHT ORGX ORGY WINDOW PROMPTMSG*) [Function]

Performs the same prompting as GETBOXPOSITION and returns the REGION specified by the user instead of the POSITION of its lower left corner.

19.14 MENUS

A menu is basically a means of selecting from a list of items. The system provides common layout and interactive user selection mechanisms, then calls a user-supplied function when a selection has been confirmed. The two major constituents of a menu are a list of items and a "when selected function." The label that appears for each item is the item itself for non-lists, or its CAR if the item is a list. The menu includes a position on the screen where it will be displayed and a means of specifying the place in the menu that is to be put at that position. In addition, there are a multitude of different formatting parameters for specifying font, size, and layout. When a menu is created, its unspecified fields are filled with defaults and its screen image is computed and saved.

Menus can be either pop up or fixed. If fixed menus are used, the menu must be included in a window.

(MENU *MENU POSITION*) [Function]

This function provides menus that pop up when they are used. It displays *MENU* at *POSITION* (in screen coordinates) and waits for the user to select an item with a mouse key. While any key is down, the selected menu item is video reversed. When all keys are released, *MENU*'s WHENSELECTEDFN field is called with three arguments: (1) the item selected, (2) the menu, and (3) the last mouse key released (LEFT, MIDDLE, or RIGHT), and MENU returns its value. If no item is selected, MENU returns NIL. If *POSITION* is NIL, the menu is brought up at the value from *MENU*'s MENUPOSITION field, if it is a POSITION, or at the current cursor position. The orientation of *MENU* with respect to the specified position is determined by its MENUOFFSET field.

(ADDMENU *MENU WINDOW POSITION* —) [Function]

This function provides menus that remain active in windows. ADDMENU displays *MENU* at *POSITION* in *WINDOW* (*POSITION* is defaulted as in MENU except that it is in window coordinates). *MENU* is added to the MENU property of *WINDOW*. The CURSORINFN and BUTTONEVENTFN of *WINDOW* are replaced with MENUBUTTONFN, so that *MENU* will be active during TTY wait. RESHAPEFN of *WINDOW* is set to restore *MENU*'s image when the window is reshaped. When an item is selected, the value of the WHENSELECTEDFN field of *MENU* is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (LEFT, MIDDLE, or RIGHT). More than one menu can be put in a window, but a menu can only be added to one window at a time. If *WINDOW* is not given, a window is created at *POSITION* (in screen coordinates) that is the size of *MENU*.

ADDMENU returns the window into which *MENU* is placed.

(DELETEMENU *MENU CLOSEFLG FROMWINDOW*) [Function]

This function removes *MENU* from the window *FROMWINDOW*. If *MENU* is the only menu in the window and *CLOSEFLG* is non-NIL, its window will be closed (by CLOSEW).

If *FROMWINDOW* is NIL, the list of currently active (open) windows is searched for one that contains *MENU*. If non is found, DELETEMENU does nothing.

19.14.1 Menu Fields

A menu is a datatype with the following fields:

ITEMS [Menu Field]

The list of items to appear in the menu. If an item is a list, its CAR will appear in the menu. If the item (or its CAR) is a bitmap, the bitmap will be displayed in the menu. The default selection functions interpret each item as a list of three elements: a label, a form whose value is returned upon selection, and a help string that is printed in the prompt window when the user presses a mouse key with the cursor pointing to this item.

WHENSELECTEDFN [Menu Field]

A function to be called when an item is selected. The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (LEFT, MIDDLE, or RIGHT). The default function DEFAULTWHENSELECTEDFN evaluates and returns the value of the CADR of the item if there is one, or simply returns the item if it is not a list or if its CADR is NIL.

WHENHELDFN [Menu Field]

The function which is called when the user has held a mouse key on an item for MENUHELDWAIT milliseconds (initially 1200). The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (LEFT, MIDDLE, or RIGHT). WHENHELDFN is intended for prompting users. The default is DEFAULTMENUHELDFN which prints (in the prompt window) the third element of the item or, if there is not a third element, the string "This item will be selected when the button is released."

WHENUNHELDFN [Menu Field]

If WHENHELDFN was called, WHENUNHELDFN will be called: (1) when the cursor leaves the item, (2) when a mouse key is released, or (3) when another key is pressed. The function is called with the same three argument values used to call WHENHELDFN. The default WHENUNHELDFN is the function CLRPROMPT (page 19.19), which just clears the prompt window.

MENUPOSITION [Menu Field]

The position of the menu to be used if the call to MENU or ADDMENU does not specify a position. For popup menus, this is in screen coordinates. For fixed menus, it is in the coordinates of the window the menu is in. The point within the menu image that is placed at this position is determined by MENUOFFSET. If MENUPOSITION is NIL, the menu will be brought up at the cursor position.

MENUOFFSET [Menu Field]

The position in the menu image that is to be located at MENUPOSITION. The default offset is (0,0). For example, to bring up a menu with the cursor over a particular menu item, set its MENUOFFSET to a position within that item and set its MENUPOSITION to NIL.

MENUFONT [Menu Field]

The font in which the items will be appear in the menu. Default is the value of

MENUFONT, initially Helvetica 10.

TITLE [Menu Field]

If specified, a title will appear in a line above the menu. The title will be in the same font as window titles.

CENTERFLG [Menu Field]

If non-NIL, the menu items are centered; otherwise they are left-justified.

MENUROWS [Menu Field]
MENUCOLUMNS [Menu Field]

These fields control the shape of the menu in terms of rows and columns. If MENUROWS is given, the menu will have that number of rows. If MENUCOLUMNS is given, the menu will have that number of columns. If only one is given, the other one will be calculated to generate the minimal rectangular menu. (Normally only one of MENUROWS or MENUCOLUMNS is given.) If neither is given, the items will be in one column.

ITEMHEIGHT [Menu Field]

The height of each item box in the menu. If not specified, it will be the maximum of the height of the MENUFONT and the heights of any bitmaps appearing as labels.

ITEMWIDTH [Menu Field]

The width of each item box in the menu. If not specified, it will be the width of the largest item in the menu.

MENUBORDERSIZE [Menu Field]

The size of the border around each item box. If not specified, 0 (no border) is used.

MENUOUTLINESIZE [Menu Field]

The size of the outline around the entire menu. If not specified, a maximum of 1 and the MENUBORDERSIZE is used.

CHANGEOFFSETFLG [Menu Field]

(popup menus only) If CHANGEOFFSETFLG is non-NIL, the position of the menu offset is set each time a selection is confirmed so that the menu will come up next time in the same position relative to the cursor. This will cause the menu to reappear in the same place on the screen if the cursor has not moved since the last selection. This is implemented by changing the MENUOFFSET field on each use. If CHANGEOFFSETFLG is the atom X or the atom Y, only the X or the Y coordinate of the MENUOFFSET field will be changed. For example, by setting the MENUOFFSET position to (-1,0) and setting CHANGEOFFSETFLG to Y, the menu will pop up so that the cursor is just to the left of the last item selected. This is the setting of the window command menus.

The following fields are read only.

IMAGEHEIGHT [Menu Field]

Returns the height of the entire menu.

IMAGEWIDTH [Menu Field]

Returns the width of the entire menu.

19.14.2 Miscellaneous Menu Functions

(WFROMMENU *MENU*) [Function]

Returns the window *MENU* is located in, if it is in one; NIL otherwise.

(DOSELECTEDITEM *MENU ITEM BUTTON*) [Function]

Calls *MENU*'s WHENSELECTEDFN on *ITEM* and *BUTTON*. It provides a programmatic way of making a selection. It does not change the display.

(MENUITEMREGION *ITEM MENU*) [Function]

Returns the region occupied by *ITEM* in *MENU*.

(SHADEITEM *ITEM MENU SHADE DSORW*) [Function]

Shades the region occupied by *ITEM* in *MENU*. If *DSORW* is a display stream or a window, it is assumed to be where *MENU* is displayed. Otherwise, WFROMMENU is called to locate the window *MENU* is in.

19.14.3 Examples of Menu Use

```
(create MENU ITEMS ← '((YES T) (NO)) )
```

Creates a menu with items YES and NO in a single vertical column. If YES is selected, T will be returned. Otherwise, NIL will be returned.

```
(create MENU ITEMS ← '(1 2 3 4 5 6 7 8 9 * 0 #)
             CENTERFLG ← T
             MENUCOLUMNS ← 3
             MENUFONT ← (FONTCREATE 'HELVETICA 10 'BOLD)
             ITEMHEIGHT ← 15
             ITEMWIDTH ← 15
             CHANGEOFFSETFLG ← T)
```

Creates a touch-tone-phone number pad with the items in 15 by 15 boxes printed in Helvetica 10 bold font. If used in pop up mode, its first use will have the cursor in the middle. Subsequent use will have the cursor in the same relative location as the previous selection.

```
(SELECTQ [MENU
          (COND ((type? MENU FOOMENU)
                     (* use previously computed menu.)
                     FOOMENU)
                (T (* create and save the menu)
                     (SETQ FOOMENU
                        (create MENU
                           ITEMS ← '((A 'A-SELECTED "prompt string for A")
                                     (B 'B-SELECTED "prompt string for B"]
         (A-SELECTED  (* if A is selected) (DOATHING))
```

```
        (B-SELECTED  (* if B is selected) (DOBTHING))
        (PROGN (* user selected outside the menu) NIL)))
```

This expression displays a pop up menu with two items, A and B, and waits for the user to select one. If A is selected, DOATHING is called. If B is selected, DOBTHING is called. If neither of these is selected, the form returns NIL.

The purpose of this example is to show some good practices to follow when using menus. First, the menu is only created once, and saved in the variable FOOMENU. This is more efficient if the menu is used more than once. Second, all of the information about the menu is kept in one place, which makes it easy to understand and edit. Third, the forms evaluated as a result of selecting something from the menu are part of the code and hence will be known to masterscope (as opposed to the situation if the forms were stored as part of the items). Fourth, the items in the menu have help strings for the user. Finally, the code is commented (always worth the trouble).

19.15 GRID FUNCTIONS

A Grid is a partitioning of an arbitrary coordinate system (hereafter referred to as the "source system") into rectangles. This subsection describes functions that operate on Grids. It includes functions to draw the outline of a Grid, to translate between positions in a source system and Grid coordinates (the coordinates of the rectangle which contains a given position), and to shade Grid rectangles. A Grid is defined by its "unit grid", a region (called a GridSpec) which is the origin rectangle of the Grid in terms of the source system. Its LEFT is the X-coordinate of the left edge of the origin rectangle, its BOTTOM is the Y-coordinate of the bottom edge of the origin rectangle, its WIDTH is the width of the grid rectangles, and its HEIGHT is the height of the grid rectangles.

(GRID *GRIDSPEC UNITSWIDE UNITSHIGH GRIDBORDER DISPLAYSTREAM GRIDSHADE*) [Function]

Outlines the grid defined by *GRIDSPEC* which is *UNITSWIDE* rectangles wide and *UNITSHIGH* rectangles high on *DISPLAYSTREAM*. Each box in the grid has a border within it that is *GRIDBORDER* points on each side; so the resulting lines in the grid are 2**GRIDBORDER* thick. If *GRIDBORDER* is the atom POINT, instead of a border the lower left point of each grid rectangle will be turned on. If *GRIDSHADE* is non-NIL, it should be a texture and the border lines will be drawn in that shade.

(SHADEGRIDBOX *X Y SHADE OPERATION GRIDSPEC GRIDBORDER DISPLAYSTREAM*) [Function]

Shades the grid rectangle (*X,Y*) of *GRIDSPEC* with texture *SHADE* using *OPERATION* on *DISPLAYSTREAM*. *GRIDBORDER* is interpreted the same as for GRID.

The following two functions map from the X,Y coordinates of the source system into the Grid X,Y coordinates:

(GRIDXCOORD *XCOORD GRIDSPEC*) [Function]

Returns the Grid X-coordinate (in the Grid specified by *GRIDSPEC*) that contains the source system X-coordinate *XCOORD*.

(GRIDYCOORD *YCOORD GRIDSPEC*) [Function]

Returns the Grid Y-coordinate (in the Grid specified by *GRIDSPEC*) that contains the source system Y-coordinate *YCOORD*.

The following two functions map from the Grid X,Y coordinates into the X,Y coordinates of the source system:

`(LEFTOFGRIDCOORD` *GRIDX GRIDSPEC*`)` [Function]

Returns the source system X-coordinate of the left edge of a Grid rectangle at Grid X-coordinate *GRIDX* (in the Grid specified by *GRIDSPEC*).

`(BOTTOMOFGRIDCOORD` *GRIDY GRIDSPEC*`)` [Function]

Returns the source system Y-coordinate of the bottom edge of a Grid rectangle at Grid Y-coordinate *GRIDY* (in the Grid specified by *GRIDSPEC*).

19.16 COLOR GRAPHICS

Note: This section describes the Interlisp-D facilities for using a color display. To use these facilities you need to have a Xerox 1100 or Xerox 1132 with a color display attached, and you must load in the LispUsers files `COLOR.DCOM` *and* `LLCOLOR.DCOM` *(automatically loaded by* `COLOR.DCOM`*).*

The color boards on the Xerox 1100 and the Xerox 1132 differ in design. The Xerox 1100 board supports 4 bits per pixel color. The Xerox 1132 supports 4 or 8 bits per pixel. All of the user's code should be written in higher level machine independent functions.

Both color boards produce an image that is 640 pixels wide by 480 pixels high. The image can be thought of as a paint-by-number painting where the number of a pixel is its value. The number of bits per pixel (4 on the Xerox 1100, 4 or 8 on the Xerox 1132) determines the number of difference colors that can be displayed at one time. When there are 4 bpp, 16 colors can be displayed at once. When there are 8 bpp, 256 colors can be displayed at once. A mapping table called a "color map" determines what color actually appears for each pixel value. A color map gives the color in terms of how much of the three primary colors (red, green and blue) displayed on the screen for each possible pixel value. In the following sections, the notions of "color map", and "color" are described.

19.16.1 Color Bitmaps

A "color bitmap" is actually just a bitmap that allows more than one bit per pixel. To test whether a bitmap *x* is a "color bitmap", use the following form:

```
(NEQ (fetch (BITMAP BITMAPBITSPERPIXEL) of x) 1)
```

Color bitmaps are created by calling `BITMAPCREATE` (page 19.4) with a *BITSPERPIXEL* argument of anything other than 1 or `NIL`. Currently, any value of *BITSPERPIXEL* except 1, 4, 8 or `NIL` (defaults to 1) will cause an error.

A 4 bit per pixel color screen bitmap uses approximately 76k of storage. There is only one such bitmap. The following function provides access to it:

`(COLORSCREENBITMAP)` [Function]

Returns the color bitmap that is being or will be displayed on the color display. This will be `NIL` if the color display has never been turned on (see `COLORDISPLAY`, page 19.47).

WHOLECOLORDISPLAY [Variable]

A global variable set to a REGION that covers the entire color display screen. Currently this is (CREATEREGION 0 0 640 480).

COLORSCREENWIDTH [Variable]

The width of the color display. Currently 640.

COLORSCREENHEIGHT [Variable]

The height of the color display. Currently 480.

19.16.2 Color Specifications

A color map maps a color number (from 0 to $2^{BITSPERPIXEL}$-1) into the intensities of the three color guns (red, green and blue). Each entry in the color map has 8 bits for each of the primary colors allowing 256 levels per primary or 2^{24} possible colors (not all of which are distinct to the human eye). Within Interlisp-D programs, colors can be manipulated as numbers, red-green-blue triples, names, or hue-lightness-saturation triples. Any function that takes a color will accept any of the different specifications.

If a number is given, it will be the color number used in the operation. It must be valid for the color bitmap used in the operation. (Since all of the routines that use a color need to determine its number, it is fastest to use numbers for colors. COLORNUMBERP described below provides a way to translate into numbers from the other representations.)

A red-green-blue (RGB) triple is a list of three numbers between 0 and 255. The first element gives the intensity for RED, the second for GREEN and the third for BLUE. When an RGB triple is used, the current color map is searched to find the color with the correct intensities. If none is found, an error is generated. (That is, no attempt is made by the system to assign color numbers to intensities automatically.) Example of an RGB triple is (255 255 255) which gives the color white. The record RGB with fields RED, GREEN, and BLUE is provided to manipulate RGB triples.

A color name is an atom that is on the association-list COLORNAMES. The CDR of the color name's entry will be used as the color corresponding to the color name. This can be any of the other representations. (Note: It can even be another color name. Loops in the name space such as would be caused by putting '(RED . CRIMSON) and '(CRIMSON . RED) on COLORNAMES are *not* checked for by the system.) Several color names are available in the initial system and are intended to allow color programs written by different users to coexist. These are:

name	RGB	number in default color map
BLACK	(0 0 0)	0
BLUE	(0 0 255)	1
GREEN	(0 255 0)	2
CYAN	(0 255 255)	3
RED	(255 0 0)	4
MAGENTA	(255 0 255)	5
YELLOW	(255 255 0)	6
WHITE	(255 255 255)	7

A hue-lightness-saturation triple is a list of three numbers. The first number (hue) is between 0 and 355 and indicates a position in degrees on a color wheel (blue at 0, red at 120 and green at 240). The second (lightness) is a FLOATP between 0 and 1 which indicates how much total intensity is in the color. The third (saturation) is a FLOATP between 0 and 1 which indicates how disparate the three primary levels are. The record HLS with fields HUE, LIGHTNESS, and SATURATION is provided to manipulate HLS triples. Example: the color blue is represented in HLS notation by (0 .5 1.0).

(COLORNUMBERP *COLOR BITSPERPIXEL NOERRFLG*) [Function]

Returns the color number (offset into the screen color map) of *COLOR*. *COLOR* should be either (1) a positive number less than the maximum number of colors, (2) a color name, (3) an RGB triple, or (4) an HLS triple. If *COLOR* is one of the above and is found in the screen colormap, its color number in the screen color map is returned. If not, an error is generated unless *NOERRFLG* is non-NIL, in which case NIL is returned.

(RGBP *x*) [Function]

Returns *x* if *x* is an RGB triple; NIL otherwise.

(HLSP *x*) [Function]

Returns *x* if *x* is an HLS triple; NIL otherwise.

19.16.3 Color Maps

The screen color map holds the information about what color is displayed on the color screen for each pixel value in the color screen bitmap. The values in the current screen color map may be changed and this change will be reflected in the colors being displayed at the next vertical retrace (approximately 1/30 of a second). Changing the color map can be used to get dramatic effects.

(COLORMAPCREATE *INTENSITIES BITSPERPIXEL*) [Function]

Creates a color map for a screen that has *BITSPERPIXEL* bits per pixel. If *BITSPERPIXEL* is NIL, the number of bits per pixel is taken from the current color display setting. *INTENSITIES* specifies the initial colors that should be in the map. If *INTENSITIES* is not NIL, it should be a list of color specifications (other than color numbers), e.g. the list of RGB triples returned by the function INTENSITIESFROMCOLORMAP (below). If *INTENSITIES* is NIL, the default is the value of \DEFAULTCOLORINTENSITIES (if *BITSPERPIXEL* is 4) or \DEFAULT8BITCOLORINTENSITIES (if *BITSPERPIXEL* is 8).

(COLORMAPP *COLORMAP? BITSPERPIXEL*) [Function]

Returns *COLORMAP?* if it is a color map that has *BITSPERPIXEL* bits per pixel; NIL otherwise. If *BITSPERPIXEL* is NIL, it returns *COLORMAP?* if it is either a 4 bits per pixel or an 8 bits per pixel colormap.

(INTENSITIESFROMCOLORMAP *COLORMAP*) [Function]

Returns a list of the intensity levels of *COLORMAP* (default is (SCREENCOLORMAP)) in a form accepted by COLORMAPCREATE. This list can be written on file and thus provides a way of saving color map specifications.

(COLORMAPCOPY *COLORMAP BITSPERPIXEL*) [Function]

If *COLORMAP* is a color map, it returns a color map that contains the same color intensities as *COLORMAP*; otherwise it returns a color map with default color values.

(SCREENCOLORMAP *NEWCOLORMAP*) [Function]

Reads and sets the color map that is used by the color display. If *NEWCOLORMAP* is non-NIL, it should be a color map and SCREENCOLORMAP sets the system color map to be that color map. Returns the previous value of the screen color map. If *NEWCOLORMAP* is NIL, the current screen color map is returned without change.

(MAPOFACOLOR *PRIMARIES*) [Function]

Returns a color map which is different shades of one or more of the primary colors. For example, (MAPOFACOLOR '(RED GREEN BLUE)) gives a color map of different shades of gray; (MAPOFACOLOR 'RED) gives different shades of red.

The following functions are provided to access and change the intensity levels in a color map.

(SETCOLORINTENSITY *COLORMAP COLORNUMBER COLORSPEC*) [Function]

Sets the primary intensities of color number *COLORNUMBER* in the color map *COLORMAP* to the ones specified by *COLORSPEC*. *COLORSPEC* can be either an RGB triple, an HLS triple or a color name. Returns NIL.

(COLORLEVEL *COLORMAP COLORNUMBER PRIMARYCOLOR NEWLEVEL*) [Function]

Sets and reads the intensity level of the primary color *PRIMARYCOLOR* (either RED, GREEN or BLUE) for the color number *COLORNUMBER* in the color map *COLORMAP*. If *NEWLEVEL* is a number between 0 and 255, it is set. The previous value of the intensity of *PRIMARYCOLOR* is returned.

(ADJUSTCOLORMAP *PRIMARYCOLOR DELTA COLORMAP*) [Function]

Adds *DELTA* to the intensity of the primary color *PRIMARYCOLOR* (either RED, GREEN or BLUE) for every color number in *COLORMAP*.

(ROTATECOLORMAP *COLORMAP STARTCOLOR THRUCOLOR*) [Function]

Rotates a sequence of colors in *COLORMAP*. The rotation moves the intensity values of color number *STARTCOLOR* into color number *STARTCOLOR*+1, the intensity values of color number *STARTCOLOR*+1 into color number *STARTCOLOR*+2, etc. and *THRUCOLOR*'s values into *STARTCOLOR*.

(EDITCOLORMAP *VAR NOQFLG*) [Function]

Allows interactive editing of a color map. If *VAR* is an atom whose value is a color map, its value is edited. Otherwise a new color map is created and edited. The color map being edited is made the screen color map while the editing is taking place so that its effects can be observed. The edited color map is returned as the value.

If *NOQFLG* is NIL and the color display is on, the user is asked if they want a test pattern of colors. A yes response will cause the function SHOWCOLORTESTPATTERN to be called which will display a test pattern with blocks of each of the possible colors.

The user is prompted for the location of a color control window to be placed on the black and white display. This window allows the value of any of the colors to be changed. The color number of the color being edited is in the upper left part of the window. Six bars are displayed. The right three bars give the color intensities for the three primary colors of the current color number. The left three bars give the value of the color's Hue, Lightness and Saturation parameters. These levels can be changed by positioning the cursor in one of the bars and pressing the LEFT button. While the LEFT button is down, the value of that parameter will track the Y position of the cursor. When the LEFT button is released, the color tracking stops. The color being edited is changed by pressing the MIDDLE button while the cursor is in the interior of the edit window. This will bring up a menu of color numbers. Selecting one sets the current color to the selected color.

The color being edited can also be changed by selecting the menu item "PickPt". This will switch the cursor onto the color screen and allow the user to select a point from the color screen. It will then edit the color of the selected point.

To stop the editing, move the cursor into the title of the editing window and press the MIDDLE button. This will bring up a menu. Select STOP to quit.

19.16.4 Turning the Color Display On and Off

The color display can be turned on and off. While the color display is on, the memory used for the color display screen bitmap is locked down and a significant amount of processing time (35% on the Xerox 1100) is used to drive the color display.

(COLORDISPLAYP) [Function]

Returns the current color map if the color display is on; otherwise NIL.

(COLORDISPLAY *COLORMAP BITSPERPIXEL CLEARSCREENFLG*) [Function]

If *COLORMAP* is NIL, it turns off the color display. If *COLORMAP* is non-NIL, it turns on the color display allocating *BITSPERPIXEL* bits per pixel. If *COLORMAP* is a color map, it is used as the screen color map. If *CLEARSCREENFLG* is non-NIL, all of the bits in the color screen are set to 0.

Turning on the color display requires allocating and locking down the memory necessary to hold the color display screen bitmap and the system color map. Turning the color display off frees this memory.

19.16.5 Printing and Drawing in Color

The current color implementation allows display streams to operate on color bitmaps. The following two functions set the color in which a display stream prints or draws:

(DSPCOLOR *COLOR DISPLAYSTREAM*) [Function]

Sets the foreground color of a display stream. Returns the previous foreground color. If *COLOR* is NIL, it returns the current foreground color without changing anything. The default foreground color is 7, which is white in the default color map.

(DSPBACKCOLOR *COLOR DISPLAYSTREAM*) [Function]

Sets the background color of a display stream. Returns the previous background color. If *COLOR* is NIL, it returns the current background color without changing anything. The default background color is 0 which is black in the default color map.

BITBLT, the line and curve drawing routines and the printing routines know how to operate on a display stream that has a color bitmap as its destination. Following are some notes about them.

BITBLT (page 19.4) When BITBLTing from a color bitmap onto another color bitmap with the same bits per pixel, the operations PAINT, INVERT and ERASE are done on a bit level; not on a pixel level. Thus painting color 3 onto color 10 will result in color 11.

When BITBLTing from a black and white bitmap onto a color bitmap, the 1 bits will appear in the DSPCOLOR and the 0 bits in DSPBACKCOLOR. Currently, REPLACE is the only operation that is supported BITBLTing from black and white to color. This operation is fairly expensive; if the same bitmap is going to be put up several times in the same color it is faster to create a color copy then blt the color copy.

If the *SOURCETYPE* is TEXTURE and the *DESTINATIONBITMAP* is a color bitmap, the *TEXTURE* argument is taken to be a color. Thus, to fill an area with the color BLUE, do:

```
(BITBLT NIL NIL NIL COLORBITMAP 50 75 100 200 'TEXTURE 'REPLACE
'BLUE)
```

Curve drawing (page 19.14)

For the functions DRAWCIRCLE, DRAWELLIPSE and DRAWCURVE, the notion of a brush has been extended to include a color. A brush can be a list of the form (*SHAPE SIZE COLOR*). A brush can also be a bitmap, which can be color bitmap.

Line drawing (page 19.13)

The line drawing functions have been extended to take another argument which is the color the line is to appear in if the destination of the display stream is a color bitmap. If the *COLOR* argument is NIL, the DSPCOLOR of the display stream is used.

Printing Printing only works (currently) in REPLACE mode. The characters will have a foreground color of DSPCOLOR and a background of DSPBACKCOLOR. The first time a character is printed in a new color, the color images corresponding to the current font are calculated and cached. Thus the first character may take a while to appear but succeeding characters print quickly.

19.16.6 Using the Cursor on the Color Screen

The cursor can be moved to the color screen. While on the color screen, the cursor is placed using XOR mode, thus with some color maps it may be hard to see. It is automatically taken down whenever an operation is performed that changes any bits on the color screen. While the cursor is on the color screen, the black and white cursor is cleared.

(CHANGECURSORSCREEN *SCREENBITMAP*) [Function]

SCREENBITMAP must be either the value of (COLORSCREENBITMAP) or the value of (SCREENBITMAP). CHANGECURSORSCREEN moves the cursor onto the specified screen. The value returned is the screen bitmap that the cursor was on before CHANGECURSORSCREEN was called.

19.16.7 Miscellaneous Color Functions

The following functions provide some common operations on color bitmaps and display streams.

(COLORFILL *REGION COLORNUMBER COLORBITMAP OPERATION*) [Function]

Fills the region *REGION* in *COLORBITMAP* with the color *COLORNUMBER*, using the operation *OPERATION*.

(COLORFILLAREA *LEFT BOTTOM WIDTH HEIGHT COLORNUMBER COLORBITMAP OPERATION*) [Function]

Fills an area in the color bitmap with a color.

(COLORIZEBITMAP *BITMAP 0COLOR 1COLOR BITSPERPIXEL*) [Function]

Creates and returns a color bitmap copying the black and white bitmap *BITMAP*. The returned color bitmap will have color number *1COLOR* in those pixels of *BITMAP* that were 1 and *0COLOR* in those pixels of *BITMAP* that were 0. This provides a way of producing a color bitmap from a black and white bitmap. Note: this is a fairly expensive operation in terms of both time and space.

19.16.8 Demonstration programs

The following functions provide some demonstrations of the color display. These are available in the Lispusers file COLORDEMO.DCOM.

(COLORDEMO) [Function]

Brings up a menu of color demonstration programs. The system will cycle through the entries on the menu automatically, allowing each to run for a small fixed amount of time (typically 40 seconds). Selecting one of the entries in the menu will cause it to start that program.

(COLORDEMO1) [Function]

Runs the Interlisp-D logo demonstration until a button is pressed then adds

COLORKINETIC. The MIDDLE button will bring up a menu that allows changing the speed of rotation or editting the color map. The LEFT button will rotate the color map in the kinetic area.

(COLORDEMO2 *SIZE*) [Function]

Puts up a test pattern of size *SIZE*, then rotates the color map. The speed of rotation of the color map is determined by the Y position of the cursor. The MIDDLE button will bring up a menu that allows editing of the color map or changing the color map to a map of different shades of one color.

(COLORKINETIC *REGION FIRSTCOLOR LASTCOLOR*) [Function]

Runs color kinetic in a region *REGION* of the color display using colors *FIRSTCOLOR* through *LASTCOLOR*.

(TUNNEL *SPEED*) [Function]

Draws a series of concentric rectangles of increasing size in increasing color numbers. *SPEED* determines the size of the rectangles. This can then be "run" by calling ROTATEIT described below.

(MINESHAFT *N OUTFLG*) [Function]

Draws a series of concentric rectangles of size *N* in increasing color numbers. *OUTFLG* determines whether the color numbers increase or decrease. This can then be "run" by calling ROTATEIT described below.

(WELL *N*) [Function]

Draws a series of concentric circles on the color screen in increasing color numbers. The circles will be of size *N*. This can then be "run" by calling ROTATEIT described below.

(SHOWCOLORTESTPATTERN *BARSIZE*) [Function]

Displays a pattern of colors on the color display. This is useful when editing a color map. The pattern has squares of the 16 possible colors layed out in two rows at the top of the screen. Colors 0 through 7 in the top row. Colors 8 through 15 in the next row. The bottom part of the screen is then layered with bars of *BARSIZE* width with the consecutive color numbers. The pattern is designed so that every color has a border with every other color (unless *BARSIZE* is too large to allow room for every color - about 20).

(ROTATEIT *BEGINCOLOR ENDCOLOR WAIT*) [Function]

Goes into an infinite loop rotating the screen color map. The colors between *BEGINCOLOR* (default 0) and *ENDCOLOR* (default maximum color) are rotated. If *WAIT* is given, (DISMISS *WAIT*) is called each time the color map is changed. This provides an easy way of "animating" screen images.

Note: The following function is available in the Lispusers file COLORPOLYGONS.DCOM.

(COLORPOLYDEMO *COLORDS*) [Function]

Runs a version of the Polygons program on the color screen.

CHAPTER 20

INTERLISP-D DISPLAY-ORIENTED TOOLS

One of the greatest strengths of Interlisp-D is the window display system. Using this system, a number of the existing Interlisp tools have been extended, and some new ones developed. This chapter describes some of these tools.

20.1 DEDIT

DEdit is a structure oriented, modeless, display based editor for objects represented as list structures, such as functions, property lists, data values, etc. DEdit is an integral part of the standard Interlisp-D environment.

20.1.1 General Comments

DEdit is designed to be the user's primary editor for programs and data. To that end, it has incorporated the interfaces of the (older) teletype oriented Interlisp editor so the two can be used interchangeably. In addition, the full power of the teletype editor, and indeed the full Interlisp system itself, is easily accessible from within DEdit.

DEdit is structure, rather than character, oriented to facilitate selecting and operating on pieces of structure as objects in their own right, rather than as collections of characters. However, for the occasional situation when character oriented editing is appropriate, DEdit provides access to the Interlisp-D text editing facilities. DEdit is modeless, in that all commands operate on previously selected arguments, rather than causing the behavior of the interface to change during argument specification.

20.1.2 Operation

DEdit is normally called through of the following functions:

`(DF` *FN*`)` [NLambda NoSpread Function]

Calls DEdit on the definition of the function *FN*.

`(DV` *VAR*`)` [NLambda NoSpread Function]

Calls DEdit on the value of the variable *VAR*.

`(DP` *NAME PROP*`)` [NLambda NoSpread Function]

Calls DEdit on the property *PROP* of the atom *NAME*. If *PROP* is not given, the whole property list of *NAME* is edited.

(DC *FILE*) [NLambda NoSpread Function]

Calls DEdit on the file commands for the file *FILE*.

DEdit is normally installed as the default editor for all editing operations, including those invoked by other subsystems, such as the Programmer's Assistant and Masterscope. DEdit provides functions EF, EV and EP (analogous to the corresponding Dx functions) for conveniently accessing the teletype editor from within a DEdit context, e.g. from under a call to DEdit or if DEdit is installed as the default editor.

The default editor may be set with EDITMODE:

(EDITMODE *NEWMODE*) [Function]

If *NEWMODE* is non-NIL, sets the default editor to be DEdit (if *NEWMODE* is DISPLAY), or the teletype editor (if *NEWMODE* is TELETYPE). Returns the previous setting.

DEdit operates by providing an alternative, plug compatible definition of EDITL (DEDITL). The normal user entries operate by redefining EDITL and then calling the corresponding Edit function (i.e., DF calls EDITF etc). Thus, the normal Edit file package, spelling correction, etc. behavior is obtained.

If Edit commands are specified in a call to DEDITL (e.g., in calls to the editor from Masterscope), DEDITL will pass those commands to EDITL, after having placed a TTY: entry on EDITMACROS which will cause DEdit to be invoked if any interaction with the user is called for. In this way, automatic edits can be made completely under program control, yet DEdit's interactive interface is available for direct user interaction.

(RESETDEDIT) [Function]

Completely reinitializes DEdit. Closes all DEdit windows, so that the user must specify the window the next time DEdit is envoked. RESETDEDIT is also used to make DEdit recognize the new values of variables such as DEDITTYPEINCOMS, when the user changes them.

20.1.3 Interactive Operation

When DEdit is called for the first time, it prompts for an edit window, which is preserved and reused for later DEdits, and pretty prints the expression to be edited therein. (Note: The pretty printer ignores user PRETTYPRINTMACROS because they do not provide enough structural information during printing to enable selection.) A standard Interlisp-D scroll bar is set up on the left edge of the window and an edit command menu, which remains active throughout the edit, on the right edge. DEdit then goes into a select, command, execute loop, during which it yields control so that background activities, such as mouse commands in other windows, continue to be performed.

20.1.3.1 Selection

Selection in a DEdit window is as follows: the LEFT button selects the object being directly pointed at; the MIDDLE button selects the containing list; and the RIGHT button extends the current selection to the lowest common ancestor of that selection and the current position. The only things that may be pointed at are atomic objects (literal atoms, numbers, etc) and parentheses, which are considered to represent the list they delimit. White space is not selectable or editable.

When a selection is made, it is pushed on a selection stack which will be the source of operands for

DEdit commands. As each new selection pushes down the selections made before it, this stack can grow arbitrarily deep, so only the top two selections on the stack are highlighted on the screen. This highlighting is done by underscoring the topmost (most recent) selection with a solid black line and the second topmost selection with a dashed line. The patterns used were chosen so that their overlappings would be both visible and distinct, since selecting a sub-part of another selection is quite common.

Because one can invoke DEdit recursively, there may be several DEdit windows active on the screen at once. This is often useful when transferring material from one object to another (as when reallocating functionality within a set of programs). Selections may be made in any active DEdit window, in any order. When there is more than one DEdit window, the edit command menu (and the type-in buffer, see below) will attach itself to the most recently opened (or current) DEdit window.

20.1.3.2 Typein

Characters may be typed at the keyboard at any time. This will create a type-in buffer window which will position itself under the current DEdit window and do a LISPXREAD (which must be terminated by a right parenthesis or a return) from the keyboard. During the read, any character editing subsystem (such as TTYIN) that is loaded can be used to do character level editing on the typein. When the read is complete, the typein will become the current selection (top of stack) and be available as an operand for the next command. Once the read is complete, objects displayed in the type-in buffer can be selected from, scrolled, or even edited, just like those in the main window.

One can also give some editing commands directly into the typein buffer. Typing control-Z will interpret the rest of the line as a teletype editor command which will be interpreted when the line is closed. Likewise, "control-S *OLD NEW*" will substitute *NEW* for *OLD* and "control-F *X*" will find the next occurrence of *X*.

20.1.3.3 Shift-Selection

Often, significant pieces of what one wishes to type can be found in an active DEdit window. To aid in transferring the keystrokes that these objects represent into the typein buffer, DEdit supports shift-selection. Whenever a selection is made in the DEdit window with the left shift key down, the selection made is not pushed on the selection stack, but is instead *unread* into the keyboard input (and hence shows up in the typein buffer). A characteristically different highlighting is used to indicate when shift (as opposed to normal) selection is taking place.

Note that shift-selection remains active even when DEdit is not. Thus one can unread particularly choice pieces of text from DEdit windows into the typescript window.

20.1.3.4 Commands

A DEdit command is invoked by selecting an item from the edit command menu. This can be done either directly, using the LEFT mouse button in the usual way, or by selecting a subcommand. Subcommands are less frequently used commands than those on the main edit command menu and are grouped together in submenus "under" the command on the main menu to which they are most closely related. For example, the teletype editor defines six commands for adding and removing parentheses (defined in terms of transformations on the underlying list structure). Of these six commands, only two (inserting and removing parentheses as a pair) are commonly used, so DEdit provides the other four as subcommands of the common two. The subcommands of a command are accessed by selecting the command from the commands menu with the `MIDDLE` button. This will bring up a menu of the subcommand options from which a choice can be made. Subcommands are flagged in the list below with the name of the top level command of which they are options.

If one has a large DEdit window, or several DEdit windows active at once, the edit command window may be far away from the area of the screen in which one is operating. To solve this problem, the DEdit command window is a "snuggle up" menu. Whenever the `TAB` key is depressed, the command window will move over to the current cursor position and stay there as long as either the `TAB` key remains down or the cursor is in the command window. Thus, one can "pull" the command window over, slide the cursor into it and then release the `TAB` key (or not) while one makes a command selection in the normal way. This eliminates a great deal of mouse movement.

Whenever a change is made, the prettyprinter reprints until the printing stablizes. As the standard pretty print algorithm is used and as it leaves no information behind on how it makes its choices, this is a somewhat heuristic process. The `Reprint` command can be used to tidy the result up if it is not, in fact, "pretty".

All commands take their operands from the selection stack, and may push a result back on. In general, the rule is to select *target* selections first and *source* selections second. Thus, a `Replace` command is done by selecting the thing to be replaced, selecting (or typing) the new material, and then buttoning the `Replace` command in the command menu. Using *TOP* to denote the topmost (most recent) element of the stack and *NXT* the second element, the DEdit commands are:

`After` [DEdit Command]

Inserts a copy of *TOP* after *NXT*.

`Before` [DEdit Command]

Inserts a copy of *TOP* before *NXT*.

`Delete` [DEdit Command]

Deletes *TOP* from the structure being edited. (A copy of) *TOP* remains on the stack and will appear, selected, in the edit buffer.

`Replace` [DEdit Command]

Replaces *NXT* with a copy of *TOP* obtained by substituting a copy of *NXT* wherever the value of the atom `EDITEMBEDTOKEN` (initially, the & character) appears in *TOP*. This provides an `MBD` facility, see Idioms below.

`Switch` [DEdit Command]

Exchanges *TOP* and *NXT* in the structure being edited.

`()` [DEdit Command]

Puts parentheses around *TOP* and *NXT* (which can, of course, be the same element).

`( in` [DEdit Command]

Subcommand of `()`. Inserts (before *TOP* (like the `LI` Edit command)

`) in` [DEdit Command]

Subcommand of `()`. Inserts) after *TOP* (like the `RI` Edit command)

() out [DEdit Command]

Removes parentheses from *TOP*.

(out [DEdit Command]

Subcommand of () out. Removes (from before *TOP* (like the LO Edit command)

) out [DEdit Command]

Subcommand of () out. Removes) from after *TOP* (like the RO Edit command)

Undo [DEdit Command]

Undoes last command.

!Undo [DEdit Command]

Subcommand of Undo. Undoes all changes since the start of this call on DEdit.

?Undo [DEdit Command]
&Undo [DEdit Command]

Subcommands of Undo. Allows selective undoing of other than the last command. Both of these commands bring up a menu of all the commands issued during this call on DEdit. When the user selects an item from this menu, the corresponding command (and if &Undo, all commands since that point) will be undone.

Find [DEdit Command]

Selects, in place of *TOP*, the first place after *TOP* which matches *NXT*. Uses the Edit subsystem's search routine, so supports the full wildcarding conventions of Edit.

Swap [DEdit Command]

Exchanges *TOP* and *NXT* on the stack, i.e. the stack is changed, the structure being edited isn't.

The following set of commands are grouped together as subcommands of Swap because they all affect the stack and the selections, rather than the structure being edited.

Center [DEdit Command]

Subcommand of Swap. Scrolls until *TOP* is visible in its window.

Clear [DEdit Command]

Subcommand of Swap. Discards all selections (i.e., "clears" the stack).

Copy [DEdit Command]

Subcommand of Swap. Puts a copy of *TOP* into the edit buffer and makes it the new *TOP*.

Pop [DEdit Command]

Subcommand of Swap. Pops *TOP* off the selection stack.

Reprint [DEdit Command]

Reprints *TOP*.

Edit [DEdit Command]

Runs DEdit on the definition of the atom *TOP* (or CAR of list *TOP*). Uses TYPESOF to determine what definitions exist for *TOP* and, if there is more than one, asks the user, via menu, which one to use. (Note: DEdit caches each subordinate edit window in the window from which it was entered, for as long as the higher window is active. Thus, multiple DEdit commands do not incur the cost of repeatedly allocating a new window.) If *TOP* is defined and is a non-list, calls INSPECT on that value. Edit also has a variety of subcommands which allow choice of editor (DEdit, Edit, TEdit, etc.) and whether to invoke that editor on the definition of *TOP* or the form itself.

EditCom [DEdit Command]

Allows one to run arbitrary Edit commands on the structure being DEdited (there are far too many of these for them all to appear on the main menu). *TOP* should be an Edit command, which will be applied to *NXT* as the current Edit expression. On return to DEdit, the (possibly changed) current Edit expression will be selected as the new *TOP*. Thus, selecting some expression, typing (R FOO BAZ), and buttoning EditCom will cause FOO to be replaced with BAZ in the expression selected.

In addition, a variety of common Edit commands are available as subcommands of EditCom. Currently, these include ?=, GETD, CL, DW, REPACK, CAP, RAISE, and LOWER.

Break [DEdit Command]

Does a BREAKIN AROUND the current expression *TOP*. (See page 10.5.)

Eval [DEdit Command]

Evaluates *TOP*, whose value is pushed onto the stack in place of *TOP*, and which will therefore appear, selected, in the edit buffer.

Exit [DEdit Command]

Exits from DEdit (equivalent to Edit OK).

OK [DEdit Command]
Stop [DEdit Command]

Subcommands of Exit. OK exits without an error; STOP exits with an error. Equivalent to the Edit commands with the same names.

20.1.3.5 Multiple Commands

It is occasionally useful to be able to give several commands at once - either because one thinks of them as a unit or because the intervening reprettyprinting is distracting. The stack architecture of DEdit makes such multiple commands easy to construct - one just pushes whatever arguments are required for the complete suite of commands one has in mind. Multiple commands are specified by holding down the CONTROL key during command selection. As long as the CONTROL key is down, commands selected will not be executed, but merely saved on a list. Finally, when a command is selected without the CONTROL key down, the command sequence is terminated with that command being the last one in the sequence.

One rarely constructs long sequences of commands in this fashion, because the feedback of being able to inspect the intermediate results is usually worthwhile. Typically, just two or three step idioms are composed in this fashion. Some common examples are given in the next section.

20.1.3.6 Idioms

As with any interactive system, there are certain common idioms on which experienced users depend heavily. Not only is discovering the idioms of a new system tiresome, but in places the designer may have assumed familiarity with one or more of them, so not knowing them can make life quite unbearable. In the case of DEdit, many of these idioms concern easy ways to achieve the effects of specific commands from the Edit system, with which many users are already familiar. The DEdit idioms described below are the result of the experience of the early users of the system and are by no means exhaustive. In addition to those that each user will develop to fit his or her own particular style, there are many more to be discovered and you are encouraged to share your discoveries.

Because of the novel argument specification technique (postfix; target first) many of the DEdit idioms are very simple, but opaque until one has absorbed the "target-source-command" way of looking at the world. Thus, one selects where typein is to go before touching the keyboard. After typing, the target will be selected second and the typein selected on top, so that an `After`, `Before` or `Replace` will have the desired effect. If the order is switched, the command will try to change the typein (which may or may not succeed), or will require tiresome Swapping or reselection. Although this discipline seems strange at first, it comes easily with practice.

Segment selection and manipulation are handled in DEdit by first making them into a sublist, so they can be handled in the usual way. Thus, if one wants to remove the three elements between `A` and `E` in the list `(A B C D E)`, one selects `B`, then `D` (either order), then makes them into a sublist with the "`()`" command (pronounced "both in"). This will leave the sublist `(B C D)` selected, so a subsequent `Delete` will remove it. This can be issued as a single "`(); Delete`" command using multiple command selection, as described above, in which case the intermediate state of `(A (B C D) E)` will not show on the screen.

Inserting a segment proceeds in a similar fashion. Once the location of the insertion is selected, the segment to be inserted is typed as a list (if it is a list of atoms, they can be typed without parentheses and the `READ` will make them into a list, as one would expect). Then, the command sequence "`After` (or `Before` or `Replace`); `() out`" (given either as a multiple command or as two separate commands) will insert the typein and splice it in by removing its parentheses.

Moving an expression to another place in the structure being edited is easily accomplished by a delete followed by an insert. Select the location where the moved expression is to go to; select the expression to be moved; then give the command sequence "`Delete; After` (or `Before` or `Replace`)". The expression will first be deleted into the edit buffer where it will remain selected. The subsequent insertion will insert it back into the structure at the selected location.

Embedding and extracting are done with the `Replace` command. Extraction is simply a special case of replacing something with a subpiece of itself: select the thing to be replaced; select the subpart that is to replace it; `Replace`. Embedding also uses `Replace`, in conjunction with the "embed token" (the value of `EDITEMBEDTOKEN`, initially the single character atom &). Thus, to embed some expression in a `PROG`, select the expression; type "`(PROG` *VARSLST* `&)`"; `Replace`.

`Switch` can also be used to generate a whole variety of complex moves and embeds. For example, switching an expression with typein not only replaces that expression with the typein, but provides a copy of the expression in the buffer, from where it can be edited or moved to somewhere else.

Finally, one can exploit the stack structure on selections to queue multiple arguments for a sequence of commands. Thus, to replace several expressions by one common replacement, select each of the expressions to be replaced (any number), then the replacing expression. Now hit the Replace command as many times as there are replacements to be done. Each Replace will pop one selection off the stack, leaving the most recently replaced expression selected. As the latter is now a copy of the original source, the next Replace will have the desired effect, and so on.

20.1.4 DEdit Parameters

There are several global variables that can be used to affect various aspects of DEdit's operation. Although most have been alluded to above, they are summarized here for reference.

EDITEMBEDTOKEN [Variable]

Initially &. Used in both DEdit and the teletype editor to indicate the special atom used as the "embed token".

DEditLinger [Variable]

Initially T. The default behavior of the topmost DEdit window is to remain active on the screen when exited. This is occasionally inconvenient for programs that call DEdit directly, so it can be made to close automatically when exited by setting this variable to NIL.

DEDITTYPEINCOMS [Variable]

Defines the control characters recognized as commands during DEdit typein. Only accessed when DEdit is initialized, so DEdit should be reinitialized with (RESETDEDIT) if this is changed.

20.2 INTERACTIVE BITMAP EDITING

One important concept of the Interlisp-D display system is the idea of a bitmap, a rectangular array of bits. While working with the display system, it is extremely useful to be able to manipulate bitmaps, textures, and character bitmaps. The following functions provide an easy-to-use interactive editing facility for various types of bitmaps.

(EDITBM *BITMAP*) [Function]

If *BITMAP* is a bitmap, it is edited. If *BITMAP* is an atom whose value is a bitmap, its value is edited. If *BITMAP* is NIL, EDITBM asks for dimensions and creates a bitmap. If *BITMAP* is a region, that portion of (SCREENBITMAP) is used. If *BITMAP* is a window, it is brought to the top and its contents edited.

EDITBM sets up the bitmap being edited in an editing window. The editing window has two major areas: a gridded edit area in the lower part of the window and a display area in the upper left part. In the edit area, the left button will add points, the middle button will erase points. The right button provides access to the normal window commands to reposition and reshape the window. The actual size bitmap is shown in the display area.

If the bitmap is too large to fit in the edit area, only a portion will be editable. This portion can be changed by scrolling both up and down in the left margin and left and right in the bottom margin. Pressing the middle button while in the display area will bring up a menu that allows global placement of the portion of the bitmap being edited. To allow more of the bitmap to be editing at once, the window can be reshaped to make it larger or the GridSize← command described below can be used to reduce the size of a bit in the edit area.

Pressing the middle button while not in either the edit area or the display area (i.e. while in the grey area in the upper right or in the title) will bring up a command menu. There are commands to stop editing, to restore the bitmap to its initial state and to clear the bitmap. Holding the middle button down over a command will result in an explanatory message being printed in the prompt window. The commands are described below:

OK — Copies the changed image into the original bitmap, stops the bitmap editor and closes the edit windows. The changes the bitmap editor makes during the interaction occur on a copy of the original bitmap. Unless the bitmap editor is exited via OK, no changes are made in the original.

Stop — Stops the bitmap editor without making any changes to the original bitmap.

Clear — Sets all or part of the bitmap to 0. Another menu will appear giving a choice between clearing the entire bitmap or just the portion that is in the edit area. The second menu also acts as a confirmation, since not selecting one of the choices on this menu results in no action being taken.

Reset — Sets all or part of the bitmap to the contents it had when EDITBM was called. As with the Clear command, another menu gives a choice between resetting the entire bitmap or just the portion that is in the edit area.

GridSize← — Allows specification of the size of the editing grid. Another menu will appear giving a choice of several sizes. If one is selected, the editing portion of the bitmap editor will be redrawn using the selected grid size, allowing more or less of the bitmap to be edited without scrolling. The original size is chosen hueristically and is typically about 8. It is particularly useful when editing large bitmaps to set the edit grid size smaller than the original.

ShowAsTile — Tesselates the current bitmap in the upper part of the window. This is useful for determining how a bitmap will look if it were made the background (using the function CHANGEBACKGROUND). Note: The tiled display will not automatically change as the bitmap changes; to update it, use the ShowAsTile command again.

Paint — Puts the current bitmap into a window and call the window PAINT command on it. The PAINT command implements drawing with various brush sizes and shapes but only on an actual sized bitmap. The PAINT mode is left by pressing the RIGHT button and selecting the QUIT command from the menu. At this point, you will be given a choice of whether or not the changes you made while in PAINT mode should be made to the current bitmap.

CURSOR← — Makes the lower left part of the bitmap become the cursor and will prompt you for the "hot spot".

The bitmap editing window can be reshaped to provide more or less room for editing. When this happens, the space allocated to the editing area will be changed to fit in the new region.

Whenever the left or middle button is down and the cursor is not in the edit area, the section of the display of the bitmap that is currently in the edit area is complemented. Pressing the left button while not in the edit region will put the lower left 16 x 16 section of the bitmap into the cursor for as long as the left button is held down.

(EDITSHADE *SHADE*) [Function]

Opens a window that allows the user to edit small textures (4 by 4) patterns. In the edit area, the left button adds bits to the shade and the middle button erases bits from the shade. The top part of the window is painted with the current texture whenever all mouse keys are released. Thus it is possible to directly compare two textures that differ by more than one pixel by holding a mouse key down until all changes are made.

If *SHADE* is a texture object, EDITSHADE starts with it, otherwise, it starts with white.

(EDITCHAR *CHARCODE FONT*) [Function]

Calls the bitmap editor (EDITBM) on the bitmap image of the character *CHARCODE* in the font *FONT*. *CHARCODE* can be a character code (as returned by CHCON1) or an atom or string, in which case the first character of *CHARCODE* is used.

20.3 DISPLAY BREAK PACKAGE

The display break package allows easier access to the information available during a break, by modifying the function BREAK1 to use the window system. It is turned on in the standard system but can be turned off with the following function:

(WBREAK *ONFLG*) [Function]

If *ONFLG* is non-NIL, installs the display break package. If *ONFLG* is NIL, it uninstalls the display break package, which makes BREAK1 behave as in Interlisp-10. WBREAK returns T if the display break package was previously installed; NIL otherwise.

The display break package maintains a trace window and as many break windows as necessary. When a break occurs, a break window is brought up near the tty window of the process that broke and the terminal stream switched to it. The title of the break window is changed to give the name of the broken function, the reason for the break, and the depth of the break recursions. If a break occurs under a previous break, a new break window is created.

While in a break window, the middle button brings up a menu of break commands (EVAL, EVAL!, EDIT, revert, ↑, OK, BT, BT!, and ?=). The commands BT and BT! bring up a backtrace menu beside the break window showing the frames on the stack. BT shows frames for which REALFRAMEP is T; BT! shows all frames. When one of the frames is selected from this menu, it is greyed and the function name and the variables bound in that frame (including local variables and PROG variables) are printed in the "backtrace frame" window. If the left button is used for the selection, only named variables are printed. If the middle button is used, all variables are printed (variables without names will appear as *var**N*). The "backtrace frame" window is an inspect window (see page 20.12). In this window, the left button can be used to select the name of the function, the names of the variables or the values of the variables.

After selecting an item, the middle button brings up a command menu of commands that apply to the selected item. If the function name is selected, a choice of editing the function or seeing the compiled code with INSPECTCODE will be given. If a variable name is selected, the command SET will be offered. Selecting SET will READ a value and set the selected to the value read. (Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. It is recommended that you exercise caution in setting variables in other than your own code.) If the item selected is a value, the inspector will be called on the selected value.

The internal break variable LASTPOS is set to the selected frame of the backtrace menu so that the normal break commands EDIT, revert, and ?= work on the currently selected frame. The commands EVAL, revert, ↑, OK, and ?= in the break menu cause the corresponding commands to be "typed in." This means that these break commands will not have the intended effect if characters have already been typed in.

The operation of the display break package is controlled by the following variables:

MaxBkMenuWidth [Variable]
MaxBkMenuHeight [Variable]

The variables MaxBkMenuWidth (default 125) and MaxBkMenuHeight (default 300) control the maximum size of the backtrace menu. If this menu is too small to contain all of the frames in the backtrace, it is made scrollable in both vertical and horizontal directions.

AUTOBACKTRACEFLG [Variable]

If the variable AUTOBACKTRACEFLG is non-NIL (default is NIL), then on error breaks the command BT is executed automatically.

BACKTRACEFONT [Variable]

The backtrace menu is printed in the font BACKTRACEFONT, which is initially Gacha 8.

CLOSEBREAKWINDOWFLG [Variable]

The system normally closes break windows after the break is exited. If CLOSEBREAKWINDOWFLG is NIL, break windows will not be closed on exit. Note: In this case, the user must close all break windows.

BREAKREGIONSPEC [Variable]

Break windows are positioned near the tty window of the broken process, as determined by the variable BREAKREGIONSPEC. The value of this variable is a region whose LEFT and BOTTOM are an offset from the LEFT and BOTTOM of the tty window. The WIDTH and HEIGHT of BREAKREGIONSPEC determine the size of the break window.

TRACEWINDOW [Variable]

The trace window, TRACEWINDOW, is used for tracing functions. It is brought up when the first tracing occurs and stays up until the user closes it. TRACEWINDOW can be set to a particular window to cause the tracing formation to print out there.

TRACEREGION [Variable]

The trace window is first created in the region TRACEREGION.

20.4 THE INSPECTOR

The Inspector provides a display-oriented facility for looking at and changing arbitrary Interlisp-D data structures. The inspector can be used to inspect all user datatypes and many system datatypes (although some objects such as numbers have no inspectable structure). The inspector displays the field names and values of an arbitrary object in a window that allows setting of the properties and further inspection of the values. This latter feature makes it possible to "walk" around all of the data structures in the system at the touch of a button. In addition, the inspector is integrated with the break package to allow inspection of any object on the stack and with the display and teletype structural editors to allow the editors to be used to "inspect" list structures and the inspector to "edit" datatypes.

The underlying mechanisms of the data inspector have been factored to allow their use as specialized editors in user applications. This functionality is described at the end of this section.

Note: Currently, the inspector does *not* have UNDOing. Also, variables whose values are changed will not be marked as such.

20.4.1 Inspect Windows

An inspect window displays two columns of values. The lefthand column lists the property names of the structure being inspected. The righthand column contains the values of the properties named on the left. For variable length data such as lists and arrays, the "property names" are numbers from 1 to the length of the inspected item and the values are the corresponding elements. For arrays, the property names are the array element numbers and the values are the corresponding elements of the array.

For large lists or arrays, or datatypes with many fields, the initial window may be too small to contain all of them. In these cases, the unseen elements can be scrolled into view (from the bottom) or the window can be reshaped to increase its size.

In an inspect window, the LEFT button is used to select things, the MIDDLE button to invoke commands that apply to the selected item. Any property or value can be selected by pointing the cursor directly at the text representing it, and clicking the LEFT button. There is one selected item per window and it is marked by having its surrounding box inverted.

The commands offered by the MIDDLE button depend on whether the selection is a property or a value. If the selected item is a value, the commands provide different ways of inspecting the selected structure. The exact commands that are given depend on the type of the value. If the value is a litatom, the commands are the types for which the atom has definitions as determined by HASDEF. Some typical commands are:

FNS	Edit the definition of the selected litatom.
VARS	Inspect the value.
PROPS	Inspect the property list.

If the value is a list, there will be choice of how to inspect the list:

Inspect	Opens an inspect window in which the properties are numbers and the values are the elements of the list.

TtyEdit	Calls the teletype structural editor on the list.

DisplayEdit	Calls the display editor on the list.

AsPList	(If the list is in P-list form) Inspects the list as a property list.

AsAList	(If the list is in ASSOC list form) Inspects the list as an association-list.

AsRecord	Brings up a submenu with all of the RECORDs in the system and inspect the list with the one chosen.

"a record type"	(If the CAR is the name of a TYPERECORD) Inspects the list as the record of the type named in its CAR.

If the value is neither a litatom or a list, the only command is Inspect, which opens an inspector window onto the selected value.

If the selected item is a property, the user will be asked for a new value and the selected property will be set to the result of evaluating the read form. The evaluation of the read form and the replacement of the selected item property will appear as their own history events and are individually undoable. Properties of system datatypes cannot be set. (There are often consistency requirements which can be inadvertently violated in ways that crash the system. This may be true of some user datatypes as well.)

20.4.2 Calling the Inspector

The inspector can be called directly, by using the function INSPECT:

(INSPECT *OBJECT ASTYPE WHERE*)	[Function]

Creates an inspect window onto *OBJECT*. If *ASTYPE* is given, it will be taken as the record type of *OBJECT*. This allows records to be inspected with their property names. If *ASTYPE* is NIL, the data type of *OBJECT* will be used to determine its property names in the inspect window.

WHERE specifies the location of the inspect window. If *WHERE* is NIL, the user will be prompted for a location. If *WHERE* is a window, it will be used as the inspect window. If *WHERE* is a region, the inspect window will be created in that region of the screen. If *WHERE* is a position, the inspect window will have its lower left corner at that position on the screen.

INSPECT returns the inspect window onto *OBJECT*, or NIL if no inspection took place.

There are several ways to open an inspect window onto an object. In addition to calling INSPECT directly, the inspector can also be called by buttoning an Inspect command inside an existing inspector window. Finally, if a non-list is edited with EDITV, the inspector is called. This also causes the inspector to be called by the Dedit command from the display editor or the EV command from the standard editor if the selected piece of structure is a non-list.

(INSPECTCODE *FN*) [Function]

Opens a window and displays the compiled code of the function *FN* using PRINTCODE. The window is scrollable.

20.4.3 Choices Before Inspection

For some datatypes there is more than one aspect that is of interest or more than one method of inspecting the object. In these cases, the inspector will bring up a menu of the possibilities and wait for the user to select one.

For litatoms, the choice includes inspecting its value, its definition, its property list, its MACRO or any other aspect returned from TYPESOF. For BITMAPs, the choice is between inspecting the bitmap's contents with the bitmap editor (EDITBM) or inspecting the bitmap's fields. For LISTPs, the choice is how to inspect it and is between a one level inspector, the teletype editor (EDITE) or the display editor (DEDIT).

20.4.4 Redisplaying an Inspect Window

An inspect window is *not* automatically updated when the structure it is inspecting is changed. The inspect window can be updated by selecting the "redisplay" command from the menu brought up by pressing the MIDDLE button in the title of the window. The "redisplay" command will cause the values of the properties to be re-fetched from the structure and redisplayed.

20.4.5 Interaction With the Display Break Package

The display break package knows about the inspector in the sense that the backtrace frame window is an inspect window onto the frame selected from the back trace menu during a break. Thus you can call the inspector on an object that is bound on the stack by selecting its frame in the back trace menu, selecting its value with the LEFT button in the back trace frame window, and selecting the inspect command with the MIDDLE button in the back trace frame window. The values of variables in frames can be set by selecting the variable name with the LEFT button and then the "Set" command with the MIDDLE button.

Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. Exercise caution in setting variables in other than your own code.

20.4.6 Controlling the Amount Displayed During Inspection

The amount of information displayed during inspection can be controlled using the following variables:

MAXINSPECTCDRLEVEL [Variable]

The inspector prints only the first MAXINSPECTCDRLEVEL elements of a long list, and will make the tail containing the unprinted elements the last item. The last item can be inspected to see further elements. Initially 50.

MAXINSPECTARRAYLEVEL [Variable]

The inspector prints only the first MAXINSPECTARRAYLEVEL elements of an array. The remaining elements can be inspected by calling the function (INSPECT/ARRAY *ARRAY BEGINOFFSET*) which inspects the *BEGINOFFSET* through the *BEGINOFFSET* + MAXINSPECTARRAYLEVEL elements of *ARRAY*. Initially 300.

INSPECTALLFIELDSFLG [Variable]

If INSPECTALLFIELDSFLG is T, the inspector will show computed fields (ACCESSFNS) as well as regular fields for structures that have a record definition. Initially T.

20.4.7 Inspect Macros

The Inspector can be extended to inspect new structures and datatypes by adding entries to the list INSPECTMACROS. An entry should be of the form (*OBJECTTYPE* . *INSPECTINFO*). *OBJECTTYPE* is used to determine the types of objects that are inspected with this macro. If *OBJECTTYPE* is a litatom, the *INSPECTINFO* will be used to inspect items whose type name is *OBJECTTYPE*. If *OBJECTTYPE* is a LIST of the form (FUNCTION *DATUM-PREDICATE*), *DATUM-PREDICATE* will be APPLYed to the item and if it returns non-NIL, the *INSPECTINFO* will be used to inspect the item.

INSPECTINFO can be one of two forms. If *INSPECTINFO* is a litatom, it should be a function that will be applied to three arguments (the item being inspected, *OBJECTTYPE*, and the value of *WHERE* passed to INSPECT) that should do the inspection. If *INSPECTINFO* is not a litatom, it should be a list of (*PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN*) where the elements of this list are the arguments for INSPECTW.CREATE, described below. From this list, the *WHERE* argument will be evaluated; the others will not. If *WHERE* is NIL, the value of *WHERE* that was passed to INSPECT will be used.

Examples:

The entry ((FUNCTION MYATOMP) PROPNAMES GETPROP PUTPROP) on INSPECTMACROS would cause all objects satisfying the predicate MYATOMP to have their properties inspected with GETPROP and PUTPROP. In this example, MYATOMP should make sure the object is a litatom.

The entry (MYDATATYPE . MYINSPECTFN) on INSPECTMACROS would cause all datatypes of type MYDATATYPE to be passed to the function MYINSPECTFN.

20.4.8 INSPECTWs

The inspector is built on the abstraction of an INSPECTW. An INSPECTW is a window with certain window properties that display an object and respond to selections of the object's parts. It is characterized by an object and its list of properties. An INSPECTW displays the object in two columns with the property names on the left and the values of those properties on the right. An INSPECTW supports the protocol that the LEFT mouse button can be used to select any property name or property value and the MIDDLE button calls a user provided function on the selected value or property. For the Inspector application, this function puts up a menu of the alternative ways of inspecting values or of the ways of setting properties. INSPECTWs are created with the following function:

(INSPECTW.CREATE *DATUM PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN*) [Function]

Creates an INSPECTW that views the object *DATUM*. If *PROPERTIES* is a LISTP, it is taken as the list of properties of *DATUM* to display. If *PROPERTIES* is an ATOM, it is APPLYed to *DATUM* and the result is used as the list of properties to display.

FETCHFN is a function of two arguments (*OBJECT PROPERTY*) that should return the value of the *PROPERTY* property of *OBJECT*. The result of this function will be printed (with PRIN2) in the INSPECTW as the value.

STOREFN is a function of three arguments (*OBJECT PROPERTY NEWVALUE*) that changes the *PROPERTY* property of *OBJECT* to *NEWVALUE*. It is used by the default *PROPCOMMANDFN* and *VALUECOMMANDFN* to change the value of a property and also by the function INSPECTW.REPLACE (described below). This can be NIL if the user provides command functions which do not call INSPECTW.REPLACE. Each replace action will be a separate event on the history list. Users are encouraged to provide UNDOable *STOREFNs*.

PROPCOMMANDFN is a function of three arguments (*PROPERTY OBJECT INSPECTW*) which gets called when the user presses the MIDDLE button and the selected item in the INSPECTW is a property name. *PROPERTY* will be the name of the selected property, *OBJECT* will be the datum being viewed, and INSPECTW will be the window. If *PROPCOMMANDFN* is a string, it will get printed in the PROMPTWINDOW when the MIDDLE button is pressed. This provides a convenient way to notify the user about disabled commands on the properties. DEFAULT.INSPECTW.PROPCOMMANDFN, the default *PROPCOMMANDFN*, will present a menu with the single command Set on it. If selected, the Set command will read a value from the user and set the selected property to the result of EVALuating this read value.

VALUECOMMANDFN is a function of four arguments (*VALUE PROPERTY OBJECT INSPECTW*) that gets called when the user presses the MIDDLE button and the selected item in the INSPECTW is a property value. *VALUE* will be the selected value (as returned by *FETCHFN*), *PROPERTY* will be the name of the property *VALUE* is the value of, *OBJECT* will be the datum being viewed, and *INSPECTW* will be the INSPECTW window. DEFAULT.INSPECTW.VALUECOMMANDFN, the default *VALUECOMMANDFN*, will present a menu of possible ways of inspecting the value and create a new Inspect window if one of the menu items is selected.

TITLECOMMANDFN is a function of two arguments (*INSPECTW OBJECT*) which gets called when the user presses the MIDDLE button and the cursor is in the title or border of the inspect window *INSPECTW*. This command function is provided so that users can implement commands that apply to the entire object. The default *TITLECOMMANDFN* (DEFAULT.INSPECTW.TITLECOMMANDFN) presents a menu with the single command Redisplay and, if it is selected, redisplays *INSPECTW* (using INSPECTW.REDISPLAY, described below).

TITLE specifies the title of the window. If *TITLE* is NIL, the title of the window will be the printed form of *DATUM* followed by the string " Inspector". If *TITLE* is the litatom DON'T, the inspect window will not have a title. If *TITLE* is any other litatom, it will be applyed to the *DATUM* and the potential inspect window (if it is known). If this result is the litatom DON'T, the inspect window will not have a title; otherwise the result will be used as a title. If *TITLE* is not a litatom, it will be used as the title.

SELECTIONFN is a function of three arguments (*PROPERTY VALUEFLG INSPECTW*) which gets called when the user releases the left button and the cursor is on one of the items. The *SELECTIONFN* allows a program to take action on the user's selection of an item in the inspect window. At the time this function is called, the selected item has been "selected". The function INSPECTW.SELECTITEM (described below) can be used to turn off this selection. *PROPERTY* will be the name of the property of the selected item.

VALUEFLG will be NIL if the selected item is the property name; T if the selected item is the property value.

WHERE indicates where the inspect window should go. Its interpretation is described in INSPECT (page 20.13).

If non-NIL, *PROPPRINTFN* is a function of two arguments (*PROPERTY DATUM*) which gets called to determine what to print in the property place for the property *PROPERTY*. If *PROPPRINTFN* returns NIL, no property name will be printed and the value will be printed to the left of the other values.

An inspect window uses the following window property names to hold information: DATUM, FETCHFN, STOREFN, PROPCOMMANDFN, VALUECOMMANDFN, SELECTIONFN, PROPPRINTFN, INSPECTWTITLE, PROPERTIES, CURRENTITEM and SELECTABLEITEMS.

(INSPECTW.REDISPLAY *INSPECTW PROPERTY* —) [Function]

Updates the display of the objects being inspected in *INSPECTW*. If *PROPERTY* is a property name or a list of property names, only those properties are updated. If *PROPERTY* is NIL, all properties are redisplayed. This function is provided because inspect windows do not automatically update their display when the object they are showing changes.

This function is called by the Redisplay command in the title command menu of an INSPECTW.

(INSPECTW.REPLACE *INSPECTW PROPERTY NEWVALUE*) [Function]

Uses the *STOREFN* of the inspect window *INSPECTW* to change the property named *PROPERTY* to the value *NEWVALUE* and updates the display of *PROPERTY*'s value in the display. This provides a functional interface for user *PROPCOMMANDFNs*.

(INSPECTW.SELECTITEM *INSPECTW PROPERTY VALUEFLG*) [Function]

Sets the selected item in an inspect window. The item is inverted on the display and put on the window property CURRENTITEM of *INSPECTW*. If *INSPECTW* has a CURRENTITEM, it is deselected. *PROPERTY* is the name of the property of the selected item. *VALUEFLG* is NIL if the selected item is the property name; T if the selected item is the property value. If *PROPERTY* is NIL, no item will be selected. (This provides a way of deselecting items.)

20.5 CHAT

CHAT is a "remote terminal" facility, that allows one to communicate with other machines while inside Interlisp-D. The function CHAT sets up a "Chat connection" to a remote machine, so that everything you type is sent to the a remote machine, and everything the remote machine prints is displayed in a "Chat window". The remote machine must support the Pup Telnet protocol.

Multiple simultaneous Chat connections are possible. To switch between typing to different Chat connections, simply button within the Chat window you want to use. CHAT prompts for a new window for each new connection, except that it saves the first window to reuse once the connection in that window is closed (other windows just go away when their connections are closed).

CHAT behaves as if its Chat window is a Datamedia-2500 terminal of the dimensions determined by the size of the window. Hence, you can talk to hosts that supply Datamedia service and expect something reasonable to happen. If the host does not pay attention to the CHAT terminal specification protocol, or you go through that host to another host, you may need to inform the host of the dimensions of your "screen"; these are given in the title bar of the chat window. The font should be Gacha10 or other fixed-width font for proper Datamedia emulation.

(CHAT *HOST LOGOPTION INITSTREAM WINDOW* —) [Function]

Opens a Chat connection to *HOST*, or to the value of DEFAULTCHATHOST. If *HOST* requires login, as determined by whether it responds to the "where is user" protocol, CHAT supplies a login sequence, or if it determines that you have a single detached job, an attach sequence. If you have more than one detached job, it simply performs a WHEREIS command for you and allows you to select the job. You may alternatively specify one of the following values for *LOGOPTION*:

LOGIN	Always perform a login.
ATTACH	Always perform an attach. This will fail if you do not have exactly one detached job.
GUEST	Login as user GUEST, password GUEST.
NONE	Do not attempt to login or attach.

If *INITSTREAM* is supplied, it is either a string or the name of a file whose contents will be read as typein. When the string/file is exhausted, input is taken from T.

If *WINDOW* is supplied, it is a window to use for the connection; otherwise, the user is prompted for a window.

While CHAT is in control, all Lisp interrupts are turned off, so that control characters can be transmitted to the remote host.

Commands can be given to an active Chat connection by bugging the MIDDLE button in the Chat window to get a command menu. Current commands are:

Close — Close this connection. Once the connection is closed, control is handed over to the main tty window. Closes the window unless this is the primary Chat window.

Suspend — Same as Close, but always leaves the window open.

New — Closes the current connection and prompts for a new host to which to open a connection in the same window.

Freeze — Hold typeout from this Chat window. Bugging the window in any way releases the hold. This is most useful if you want to switch to another, overlapping window and there is typeout in this window that would compete for screen space.

Dribble — Open a typescript file for this Chat connection (closing any previous dribble file for the window). The user is prompted for a file name; a name of NIL just closes the old dribble file.

Input — Prompts for a file to take input from. When the end of the file is reached, input reverts to T.

Clear Clears the window and resets the simulated terminal to its default state. This is useful if undesired terminal commands have been received from the remote host that place the simulated terminal into a funny state.

In an inactive Chat window, the MIDDLE button brings up a menu of one item, ReConnect, whose selection reopens a connection to the same host as was last in the window. This is the primary motivation for the Suspend menu command. A new Chat connection can also be opened from the Background menu.

The mouse button LEFT, when inside CHAT, holds output as long as the button is down. Holding down MIDDLE coincidentally does this, too, but not on purpose: since the menu handler does not yield control to other processes, it is possible to kill the connection by keeping the menu up too long.

Chat windows are a little bit knowledgable about window operations. If you reshape a Chat window, Chat informs your partner of the new dimensions. And if you close the window, the connection is also closed.

The following variables control aspects of Chat's behavior:

CHAT.DISPLAYTYPE [Variable]

The type of display (a number) that Chat should tell the remote host the user is on. If Datamedia emulation is desired, this variable should be set to the number corresponding to the terminal type Datamedia for the remote host. If the remote host does not respond to the terminal type protocol in Pup Telnet, this variable is irrelevant.

CHAT.ALLHOSTS [Variable]

A list of host names, as uppercase litatoms, that the user desires to Chat to. Chatting to a host not on the list adds it to the list. These names are placed in the menu that the background Chat command prompts with.

CLOSECHATWINDOWFLG [Variable]

If true, every Chat window is closed on exit. If NIL, the initial setting, then the primary Chat window is not closed.

DEFAULTCHATHOST [Variable]

The host to which CHAT connects when it is called with no *HOST* argument.

CHAT.FONT [Variable]

If non-NIL, the font that Chat windows are created with. If CHAT.FONT is NIL, Chat windows are created with (DEFAULTFONT 'DISPLAY).

20.6 THE TEDIT TEXT EDITOR

TEdit is a window-based, modeless text editor, capable of handling fonts and some rudimentary formatting. Text is selected with the mouse, and all editor operations act on the current selection.

The top-level entry to TEdit is:

(TEDIT *TEXT WINDOW DONTSPAWN PROPS*) [Function]

TEXT may be a (litatom) file name, an open STREAM, a string, or an arbitrary [MKSTRING-able] Lisp object. The text is displayed in an editing window, and may be edited there. If *TEXT* is other than a file name, a STREAM, or a string, TEDIT will call MKSTRING on it, and let you edit the result.

If *WINDOW* is NIL, you will be prompted to create a window. If *WINDOW* is non-NIL, TEDIT will use it as the window to edit in. If *WINDOW* has a title, TEDIT will preserve it; otherwise, TEDIT will provide a descriptive title for the window.

TEDIT will normally spawn a new process to run the edit, so you can edit in parallel with other work; indeed, it is possible to have several editing windows active on the screen. To prevent a new process from being created, call TEDIT with *DONTSPAWN* set to T.

PROPS is a prop-list-like collection of properties which control the editing session. The following options are possible:

FONT
The default font to be used in the edit window.

QUITFN
A function to call when the user Quits.

LOOPFN
A function to be called each time thru the character-read loop.

CHARFN
A function to be called for each character typed in.

SELFN
A function to be called each time a mouse selection is made in this edit window.

TERMSA
If you want characters displayed other than TEdit's default way, set this to a character table.

READONLY
If this atom is present *anywhere* in the list of *PROPS*, then the edit window will be read-only, i.e., you can only shift-select in it.

SEL
Tells what text should be selected initially. This can be a SELECTION (see below) describing the selected text, or a character number, or a two-element list of first character number and number of characters to select.

MENU
Describes the menu to be displayed when the MIDDLE mouse button is pressed in the edit window's title region. If it is a MENU, that menu will appear. If it is a list of menu items, a new menu will be constructed.

AFTERQUITFN
A function to be called *after* TEdit has quit. This can be used for cleanup of side-effects by TEdit client programs.

REGION	A window-relative region; TEdit will use only that portion of the window to display text &c. This is for people who want TEdit for filling in forms, etc.
TITLEMENUFN	A function to get called instead of bringing up the usual TEdit command menu when the user LEFT- or MIDDLE-buttons in the edit window's title region.

20.6.1 Selecting Text

TEdit works by operating on "selected" pieces of text. Selected text is highlighted in some way, and may have a caret flashing at one end. Insertions go where the caret is; deletion and other operations are applied to the currently selected text.

Text is selected using the mouse. There are two regions within an edit window: The area containing text, and a "line bar" just inside the left edge of the window. While the mouse is inside the text region, the cursor is the normal up-and-left pointing arrow. When the cursor moves into the line bar, it changes to an up-and-right pointing arrow. Which region the mouse is in determines what kind of selection happens:

The LEFT mouse button always selects the smallest things. In the text region, it selects the character you're pointing at; in the line bar, it selects the single line you're pointing at.

The MIDDLE mouse button selects larger things. In the text region, it selects the word the cursor is over, and in the line bar it selects the paragraph the cursor is next to.

The RIGHT button always extends a selection. The current selection is extended to include the character/word/line/paragraph you are now pointing at. For example, if the existing selection was a whole-word selection, the extended selection will also consist of whole words.

There are special ways of selecting text which carry an implicit command with them:

If you hold the CTRL key down while selecting text, the text will be shown white-on-black. When you release the CTRL key, the selected text will be deleted. You can abort a CTRL-selection: Hold down a mouse button, and release the CTRL key. Then release the mouse button.

Holding the SHIFT key down while making a selection causes it to be a "copy-source" selection. A copy source is marked with a dashed underline. Whatever is selected as a copy source when the SHIFT key is released will be copied to where the caret is. This even works to copy text from one edit window to another. You can abort a copy: Hold down a mouse button, and release the SHIFT key. Then release the mouse button.

Holding the CTRL and SHIFT keys down while making a selection causes it to be a "move" selection, which is marked by making it veverse video. Whatever is selected as a "move" source when the CTRL and SHIFT keys are released will be moved to where the caret is. This even works to move text from one edit window to another. You can abort a move: Hold down a mouse button, and release the CTRL and SHIFT keys. Then release the mouse button. If the variable TEDIT.BLUE.PENDING.DELETE is non-NIL, extending a selection will display the selection as white-on-black. The next time something is typed, the selected text will be deleted first.

20.6.2 Editing Operations

Inserting text: Except for command characters, whatever is typed on the keyboard gets inserted where the caret is. The `BS` key and control-A both act as a backspace, deleting the character just before the caret. Control-W is the backspace-word command.

Deleting Text: Hitting the `DEL` key causes the currently-selected text to be deleted. Alternatively, you can use the `CTRL`-selection method described above.

Copying Text: Use `SHIFT`-selection, as described above.

Moving Text: Use `CTRL-SHIFT`-selection.

Undoing an edit operation: The top blank key is the **Undo** key. It will undo the most recent edit command. **Undo** is itself undo-able, so you can never back up more than a single command.

Redoing an edit operation: The `ESC` key is the **Redo** key. It will redo the most recent edit command on the current selection. For example, if you insert some text, then select elsewhere, hitting `ESC` will insert a copy of the text in the new place also. If the last command was a delete, **Redo** will delete the currently-selected text; if it was a font change, the same change will be applied to the current selection.

The command menu: You can get command menus by moving into the edit window's title region and hitting the `RIGHT` or `MIDDLE` mouse buttons. `RIGHT` gets the usual menu of window commands. `MIDDLE` gets a menu of editor commands:

Put Causes an updated version of the file to be written. Tedit will ask you for a file name, offering the existing name (if any) as the default.

Get Lets you read in a new file to edit, *without saving the one you were working on.* You'll be asked for a file name in the prompt window.

Include Lets you copy the contents of a file into the edit window, inserting it where the caret is.

Quit Causes the editor to stop without updating the file you're editing. If you haven't saved your changes, you'll be asked to confirm this.

Find Asks for a search string, then hunts from the caret toward the end of document for a match. Selects the first match found; if there is none, nothing happens.

Substitute Asks for a search string and a replacement string. Within the current selection, all instances of the search string ware replaced by the replacement string. If you wish, TEdit will ask you to confirm each replacement before actually doing it.

Looks Changes the character looks of the selected characters: The font, character size, and face (bold, italic, etc.). Three menus will pop up in sequence: One to select the font name, one to select the face, and one to select the size. You may select an option in each menu. If, for example, you want to leave the character size alone, just click the mouse outside the size menu. In general, any aspect of the character looks that you don't change will remain the same.

Hardcopy Prints the document to your default press or InterPress printer, with 1 inch margins all around. The function PRINTERMODE controls which kind of printer TEdit will send to.

Press File — Creates a Press or InterPress file of the document, with 1 inch margins all around. The file format is also controlled by PRINTERMODE.

20.6.3 TEdit Functional Interface

The Text Stream

TEdit keeps a STREAM which describes the current state of the text you're editing. You can use most of the usual stream operations on that stream: BIN, SETFILEPTR, GETFILEPTR, GETEOFPTR, BACKBIN, and PEEKBIN do the usual things. BOUT inserts a character in the stream just in front of the next character you'd read if you BINned. You can get the stream by (WINDOWPROP *Edit-WINDOW* 'TEXTSTREAM).

If you need to save the state of an edit, you can save this stream. Calling TEDIT with the stream as the *TEXT* argument will let you continue from where you left off.

The "Text Object"

TEdit keeps a variety of other information about each edit window, in a data structure called a TEXTOBJ. Field F3 of a text STREAM points to the associated TEXTOBJ, which contains these fields of interest:

\WINDOW — The edit window which contains the text. If this is NIL, there is no edit window for this text.

SEL — The most recent selection made in this text.

SCRATCHSEL — A scratch SELECTION, used by the mouse handler for the edit window, but otherwise available for scratch use.

TEXTLEN — The current length of the edited text.

STREAMHINT — Points to the text STREAM which describes the text.

EDITFINISHEDFLG — If this is non-NIL, TEdit will halt after the next time through the keyboard polling loop. No check will be made for unsaved changes. Unless it it T, the value of EDITFINISHEDFLG will be returned as the result of TEdit.

Selections

The selected text is described by an object of type SELECTION, whose fields are as follows:

CH# — The character number of the first character in the selection. The first character in the text being edited is numbered 1.

CHLIM — The character number of the last character in the selection. Must be $\geq$ CH#.

DCH — The number of characters in the selection. If DCH is zero, then no characters are selected, and the Selection can be used only to describe a place to insert text.

ONFLG Tells whether the Selection is indicated in the edit window. If T, it is; if NIL, it's not.

\TEXTOBJ The TEXTOBJ that describes the selected text. You can use this to get to the Stream itself.

X0 The X position (edit-window-relative) of the left edge of the first selected character.

Y0 The Y position of the bottom of the first selected character (not the character's base line, the bottom of its descent).

XLIM The X position of the right edge of the last character selected. If DCH is zero (a "point" selection), XLIM=X0.

YLIM The bottom of the last character in the selection.

DX The width of the selection. If DCH is zero, this will be also.

SELOBJ This is for a future object-oriented editing interface.

POINT Tells which side of the selection the caret should appear on. It will be one of the atoms LEFT and RIGHT.

SET T if this selection is currently valid, NIL if it is obsolete or has never been set.

SELKIND What kind of selection this is. One of the atoms CHAR, WORD, LINE, or PARA.

HOW A TEXTURE, which will be used to highlight the selecton.

HOWHEIGHT How high the highlighting is to extend. A selection's highlight starts at the bottom of the lowest descender, and extends upward for HOWHEIGHT pixels. To always get highlighting a full line tall, set this to 16384.

HASCARET T if this selection should have a caret flashing next to it, NIL otherwise.

20.6.3.1 TEdit Interface Functions

TEdit exports the following functions for use in custom interfaces:

(OPENTEXTSTREAM *TEXT WINDOW START END PROPS*) [Function]

Creates a text STREAM describing *TEXT*, and returns it. If *WINDOW* is specified, the text will be displayed there, and any changes to the text will be reflected there as they happen. You will also be able to scroll the window and select things there as usual. *TEXT* may be an existing TEXTOBJ or text STREAM. If *START* and *END* are given, then only the section of *TEXT* delimited is edited. *PROPS* is the same as for TEDIT.

Given the STREAM, you can use a number of functions to change the text in an edit window, under program control. The edit window gets updated as the text is changed.

(TEDIT.SETSEL *STREAM CH#orSEL LEN POINT*) [Function]

Sets the selection in *STREAM*. If *CH#orSEL* is a SELECTION, it is used as-is. Otherwise, *CH#orSEL* is the first character in the selection, and *LEN* is the number of characters to select (zero is allowed, and gives just an insertion point). *POINT* tells which side of the selection the caret should come on. It must be one of the atoms LEFT or RIGHT.

(TEDIT.GETSEL *STREAM*) [Function]

Returns the SELECTION which describes the current selection in the edit window described by *STREAM*.

(TEDIT.SHOWSEL *STREAM ONFLG SEL*) [Function]

Lets you turn the highlighting of the selection *SEL* on and off. If *ONFLG* is T, the selection *SEL* in *STREAM* will be highlit in the edit window; if NIL, any highlighting will be turned off. If *SEL* is NIL, it defaults to the current selection in *STREAM*.

(TEDIT.INSERT *STREAM TEXT CH#orSEL*) [Function]

Inserts the string *TEXT* into *STREAM*, as though it had been typed in. *CH#orSEL* tells where to insert the text: If it's NIL, the text goes in where the caret is. If it's a FIXP, the text is inserted in front of the corresponding character (The first character in the stream is numbered 1). If it's a SELECTION, the text is inserted accordingly.

(TEDIT.DELETE *STREAM CH#orSEL LEN*) [Function]

Deletes text from *STREAM*. If *CH#orSEL* is a SELECTION, the text it describes will be deleted; if *CH#orSEL* is a FIXP, it is the character number of the first character to delete. In that case, *LEN* must also be present; it is the number of characters to be deleted.

(TEDIT.FIND *STREAM TEXT CH#*) [Function]

Searches for the next occurence of *TEXT* inside *STREAM*. If *CH#* is present, the search starts there; otherwise, the search starts from the caret. If it finds a match, TEDIT.FIND returns the character number of the first character in the matching text. If no match is found, it returns NIL.

(TEDIT.HARDCOPY *STREAM FILE DONTSEND BREAKPAGETITLE*) [Function]

Sends the text contained in *STREAM* to the printer. If a file name is given in *FILE*, the press file will be left there for you to use. If *DONTSEND* is non-NIL, the file will not be sent to the printer; use this if you only want to create a press file for later use.

If *BREAKPAGETITLE* is non-NIL, it is used as the title on the "break page" printed before the text.

(TEDIT.LOOKS *STREAM NEWLOOKS SELORCH# LEN*) [Function]

Changes the character looks of selected characters, e.g., the font, character size, etc. *SELORCH#* can be a SELECTION, an integer, or NIL. If *SELORCH#* is a SELECTION, the text it describes will be changed; if it is a FIXP, it is the character number of the first character to changed. In that case, *LEN* must also be present; it is the number of characters to be changed.

NEWLOOKS is a property-list-like description of the changes to be made. The property names tell what to change, and the property values describe the change. Any property which isn't changed explicitly retains its old value. Thus, it is possible to make a piece of text all bold without changing the fonts the text is in. The possible list entries are as follows:

FAMILY The name of the font family. All the selected text is changed to be in that font.

FACE The face for the new font. This may be in either of the two forms acceptable to FONTCREATE: a list such as (BOLD ITALIC REGULAR), or an atom such as MRR.

SIZE The new point size.

UNDERLINE The value for this property must be one of the atoms ON or OFF. The text will be underscored or not, accordingly.

OVERLINE The value for this property must be one of the atoms ON or OFF. The text will be overscored or not, accordingly.

STRIKEOUT The value for this property must be one of the atoms ON or OFF. The text will be struck through with a single line or not, accordingly.

SUPERSCRIPT A distance, in points. The text will be raised above the normal baseline by that amount. This is mutually exclusive with SUBSCRIPT.

SUBSCRIPT A distance, in points. The text will be raised above the normal baseline by that amount. This is mutually exclusive with SUPERSCRIPT.

PROTECTED The value for this property must be one of the atoms ON or OFF. If it is ON, the text will be protected from mouse selection and from deletion.

SELECTPOINT The value for this property must be one of the atoms ON or OFF. If a character has this property, the user can make a point selection just after it, even if the character is also PROTECTED.

(TEDIT.QUIT *STREAM VALUE*) [Function]

STREAM must be the text stream associated with a running TEdit. TEDIT.QUIT causes the editing session to end. If *VALUE* is given, it is returned as TEdit's result; otherwise, TEdit will return the usual result. The user is not asked to confirm his desire to stop editing.

(TEDIT.ADD.MENUITEM *MENU ITEM*) [Function]

Adds a menu *ITEM* to *MENU*. This will update the menu's image so that the newly-added item will appear the next time the menu pops up. This is only guaranteed to work right with pop-up menus which aren't visible.

(TEDIT.REMOVE.MENUITEM *MENU ITEM*) [Function]

Removes a menu *ITEM* from *MENU*. This will update the menu's image so that the newly-added item will appear the next time the menu pops up. This is only guaranteed to work right with pop-up menus which aren't visible. *ITEM* may be either the whole menu item, or just the indicator which appears in the menu's image.

20.6.3.2 User-function "Hooks" in TEdit

TEdit provides a number of hooks where a user-supplied function can be called. To supply a function, attach it to the edit window under the appropriate indicator, using WINDOWPROP. Every user-supplied function is APPLYed to the text STREAM which describes the text. Some of these functions can also be supplied using the *PROPS* argument to TEDIT or OPENTEXTSTREAM; the descriptions below contain the details.

TEDIT.QUITFN [Window Property]

A function to be called whenever the user ends an editing session. This may do anything; if it returns the atom DON'T, TEdit will not terminate. Any other result permits TEdit to do its normal cleanup and termination. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.AFTERQUITFN [Window Property]

A function to be called after the user ends an editing session. This may perform any cleanup of side effects that you desire. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.CMD.LOOPFN [Window Property]

A function that gets called, for effect only, each time through TEdit's main command loop. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.CMD.CHARFN [Window Property]

A function that gets called, for effect only, once for each character typed into TEdit. The character code is passed to the function as its second argument. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.CMD.SELFN [Window Property]

A function that gets called, for effect only, each time the user selects something with the mouse. The new SELECTION is passed as the function's second argument, and an atom describing the kind of selection (one of NORMAL, COPY, MOVE, or DELETE) as the third. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.PRESCROLLFN [Window Property]

Called just before TEdit scrolls the edit window.

TEDIT.POSTSCROLLFN [Window Property]

Called just after TEdit scrolls the edit window.

TEDIT.OVERFLOWFN [Window Property]

Called when TEdit is about to move some text off-screen. This function may handle the text overflow itself (say by reshaping the window), or it may let TEdit take its normal course. If the function handles the problem, it must return a non-NIL result. If TEdit is to handle the overflow, the value returned must be NIL.

TEDIT.TITLEMENUFN [Window Property]

Called whenever the user presses the LEFT or MIDDLE mouse button in the edit window's title region. Can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM. Normally, this is the function TEDIT.DEFAULT.MENUFN, which brings up the usual TEdit command menu.

TEdit also saves pointers to its data structures on each edit window. They are available for any user function's use.

TEXTOBJ [Window Property]

The TEXTOBJ which describes the current editing session.

TEXTSTREAM [Window Property]

The text STREAM which describes the text of the document.

20.6.3.3 Changing the TEdit Command Menu

You may replace the MIDDLE-button command menu with one of your own. When you press the MIDDLE button inside an edit window's title region, TEDIT calls the value of the TEDIT.TITLMENUFN window property with the window as its argument. Normally, what gets called is TEDIT.DEFAULT.MENUFN, but you may change it to anything you like.

TEDIT.DEFAULT.MENUFN brings up a menu of commands. If the edit window has a property TEDIT.MENU, that menu is used. If not, TEdit looks for the window property TEDIT.MENU.COMMANDS (a list of menu items) and constructs a menu from that. Failing that, it uses TEDIT.DEFAULT.MENU.

This means that you can control the command menu by setting the appropriate window properties. Alternatively, you may add your own menu buttons to the default menu, TEDIT.DEFAULT.MENU.

(TEDIT.ADD.MENUITEM TEDIT.DEFAULT.MENU *ITEM*)

will add *ITEM* to the TEdit menu. Menu items should be in the form (*NAME FUNCTION*), where *NAME* is what appears in the menu, and *FUNCTION* will be applied to the text stream, and can perform any operation you desire.

Finally, you may *remove* menu items from the default menu, by doing

(TEDIT.REMOVE.MENUITEM TEDIT.DEFAULT.MENU *ITEM*)

ITEM can be either a complete menu item, or just the text that appears in the menu; either will do the job.

20.6.3.4 Variables Which Control TEdit

There are a number of global variables which control TEdit, or which contain state information for editing sessions in progress:

`TEDIT.BLUE.PENDING.DELETE` [Variable]

If this is non-`NIL`, extending a selection makes it into a pending-delete selection. See the selection section.

`TEDIT.DEFAULT.FONT` [Variable]

A `FONTDESCRIPTOR`. This is the font for displaying TEdit documents which don't specify their own font information.

`TEDIT.DEFAULT.FMTSPEC` [Variable]

A paragraph-looks description. This contains the default looks for a paragraph.

`TEDIT.SELECTION` [Variable]

A `SELECTION`. This is the most recent regular selection made in *any* TEdit window.

`TEDIT.SHIFTEDSELECTION` [Variable]

A `SELECTION`. This is the most recent `SHIFT`-selection made in *any* TEdit window.

`TEDIT.MOVESELECTION` [Variable]

A `SELECTION`. This is the most recent `CTRL-SHIFT`-selection made in *any* TEdit window.

`TEDIT.READTABLE` [Variable]

A read table, this is used to translate typed-in characters into TEdit commands. See the section on TEdit readtables.

`TEDIT.WORDBOUND.READTABLE` [Variable]

The read table which controls TEdit's concept of word boundaries. The syntax classes in this table aslo determine which characters TEdit thinks are white space (which gets deleted by control-W along with the preceding word).

20.6.4 TEdit's Terminal Table and Readtables

TEdit now pays attention to the system terminal table. Characters with terminal sytax-classes `CHARDELETE`, `WORDDELETE`, or `LINEDELETE` act as follows:

`CHARDELETE`	acts as a character-backspace.
`WORDDELETE`	acts like control-W (in fact, this is how control-W is implemented.)
`LINEDELETE`	acts like DEL.

Since the system terminal table is used to implement these functions, you can assign them to other keys at will.

TEdit also has a Readtable, which it uses to dispatch to commands. The table is named `TEDIT.READTABLE`, and it is global. You can use the functions `TEDIT.SETSYNTAX` and `TEDIT.GETSYNTAX` to read it and make changes:

`(TEDIT.SETSYNTAX` *`CHARCODE CLASS TABLE`*`)` [Function]

Sets the readtable syntax of the character whose charcode is *`CHARCODE`* to be

CLASS in the read-table *TABLE*. The possible syntax classes are listed below.

(TEDIT.GETSYNTAX *CHARCODE TABLE*) [Function]

Returns the TEdit syntax class of the character whose charcode is *CHARCODE*, according to the read-table *TABLE*. The possible syntax classes are listed below. An illegal syntax will be returned as NIL.

The allowable syntax classes are:

CHARDELETE Typing this character acts like backspace

WORDDELETE Typing this character acts like controlW

DELETE Typing this character acts like DEL

UNDO Typing this character causes Undo

REDO Typing this character acts like ESC

FN Typing this character calls a specified function (see below)

NONE Typing this character simply inserts it in the document. NIL also has this effect.

You can also cause a keystroke to invoke a function for you. To do so, use the function

(TEDIT.SETFUNCTION *CHARCODE FN TABLE*) [Function]

Sets up the TEdit readtable *TABLE* so that typing the character with charcode *CHARCODE* will APPLY *FN* to the text STREAM and the TEXTOBJ for the document being edited. The function may have arbitrary side-effects.

The abbreviation feature described below is implemented using this function-call facility.

Finally, TEdit uses the read table TEDIT.WORDBOUND.READTABLE to decide where word boundaries are. Whenever two adjacent characters have different syntax classes, there is a word boundary between them. The state of this table can be controlled by the functions

(TEDIT.WORDGET *CHAR TABLE*) [Function]

Returns the syntax class (a small integer) for a given character. *CHAR* may be either a character or a charcode; *TABLE* defaults to TEDIT.WORDBOUND.READTABLE.

(TEDIT.WORDSET *CHAR CLASS TABLE*) [Function]

Sets the syntax class for a character. Again, *CHAR* is either a character or a charcode; *TABLE* defaults to TEDIT.WORDBOUND.READTABLE; *CLASS* may be either a small integer as returned by TEDIT.WORDGET, or one of the atoms WHITESPACE, TEXT, or PUNCTUATION. Those represent the syntax classes in the default TEDIT.WORDBOUND.READTABLE.

The initial TEDIT.WORDBOUND.READTABLE assigns every character to one of the above classes, along pretty obvious lines. For purposes of control-W, whitespace between the caret and the word being deleted is also removed.

20.6.5 The TEdit Abbreviation Facility

The list TEDIT.ABBREVS is a list of "abbreviations known to TEdit." Each element of the list is a dotted pair of two strings. The first is the abbreviation (case does matter), and the second is what the abbreviation expands to. To expand an abbreviation, select it and type control-X. It will be replaced by its expansion.

You can also expand single-character abbreviations while typing. Hitting control-X when no characters are underlined (i.e., after you have typed something) will expand the *single-character* abbreviation to the left of the caret.

Here is a list of the default abbreviations and their expansions:

b	The bullet (•)
m	The M-dash (—)
n	The figure dash (–)
"	Open double-quotes (") which can be matched by two normal quotes (")

20.7 THE TTYIN DISPLAY TYPEIN EDITOR

TTYIN is an Interlisp function for reading input from the terminal. It features altmode completion, spelling correction, help facility, and fancy editing, and can also serve as a glorified free text input function. This document is divided into two major sections: how to use TTYIN from the user's point of view, and from the programmer's.

TTYIN exists in implementations for Interlisp-10 and Interlisp-D. The two are substantially compatible, but the capabilities of the two systems differ (Interlisp-D has a more powerful display and allows greater access to the system primitives needed to control it effectively; it also has a mouse, greatly reducing the need for keyboard-oriented editing commands). Descriptions of both are included in this document for completeness, but Interlisp-D users may find large sections irrelevant.

20.7.1 Entering Input With TTYIN

There are two major ways of using TTYIN: (1) set LISPXREADFN to TTYIN, so the LISPX executive uses it to obtain input, and (2) call TTYIN from within a program to gather text input. Mostly the same rules apply to both; places where it makes a difference are mentioned below.

The following characters may be used to edit your input, independent of what kind of terminal you are on. The more TTYIN knows about your terminal, of course, the nicer some of these will behave. Some functions are performed by one of several characters; any character that you happen to have assigned as an interrupt character will, of couse, not be read by TTYIN. There is a (somewhat inelegant) way of changing which characters perform which functions, described under TTYINREADMACROS later on.

control-A, BS, DEL

Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.

control-W
Deletes a "word". Generally this means back to the last space or parenthesis.

control-Q (control-U for Tops20 users)
Deletes the current line, or if the current line is blank, deletes the previous line.

control-R
Refreshes the current line. Two in a row refreshes the whole buffer (when doing multi-line input).

ESC Tries to complete the current word from the spelling list provided to TTYIN, if any. In the case of ambiguity, completes as far as is uniquely determined, or rings the bell. For LISPX input, the spelling list may be USERWORDS (see discussion of TTYINCOMPLETEFLG, page 20.44).

Interlisp-10 only: If no spelling list was provided, but the word begins with a "<", tries directory name completion (or filename completion if there is already a matching ">" in the current word).

? If typed in the middle of a word will supply alternative completions from the *SPLST* argument to TTYIN (if any). ?ACTIVATEFLG (page 20.43) must be true to enable this feature.

control-F Sumex, Tops20 only: Invokes GTJFN for filename completion on the current "word".

control-Y
Escapes to a Lisp userexec, from which you may return by the command OK. However, when in READ mode and the buffer is non-empty, control-Y is treated as Lisp's unquote macro instead, so you have to use edit-control-Y (below) to invoke the userexec.

<middle-blank> in Interlisp-D, LF in Interlisp-10
Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed at TTYIN; when typed in the middle of a line fills in the remaining text from the old line; when typed following ↑Q or ↑W restores what those commands erased.

; If typed as the first character of the line means the line is a comment; it is ignored, and TTYIN loops back for more input.

control-X
Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced, beeps if not. Currently implemented in Interlisp-D only.

During most kinds of input, TTYIN is in "autofill" mode: if a space is typed near the right margin, a carriage return is simulated to start a new line. In fact, on cursor-addressable displays, lines are always broken, if possible, so that no word straddles the end of the line. The "pseudo-carriage return" ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You won't get carriage returns in your strings unless you explicitly type them.

20.7.2 Mouse Commands [Interlisp-D Only]

The mouse buttons are interpreted as follows during TTYIN input:

LEFT Moves the caret tò where the cursor is pointing. As you hold down LEFT, the caret moves around with the cursor; after you let up, any typein will be inserted at the new position.

MIDDLE Like LEFT, but moves only to word boundaries.

RIGHT Deletes text from the caret to the cursor, either forward or backward. While you hold down RIGHT, the text to be deleted is complemented; when you let up, the text actually goes away. If you let up outside the scope of the text, nothing is killed (this is how to "cancel" the command). This is roughly the same as CTRL-RIGHT with no initial selection (below).

If you hold down CTRL and/or SHIFT while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. You make a selection by bugging LEFT (to select a character) or MIDDLE (to select a word), and optionally extend the selection either left or right using RIGHT. While you are doing this, the caret does not move, but your selected text is highlighted in a manner indicating what is about to happen. When you have made your selection (all mouse buttons up now), lift up on CTRL and/or SHIFT and the action you have selected will occur, which is:

SHIFT The selected text as typein at the caret. The text is highlighted with a broken underline during selection.

CTRL Delete the selected text. The text is complemented during selection.

CTRL-SHIFT
Combines the above: delete the selected text and insert it at the caret. This is how you move text about.

You can cancel a selection in progress by pressing LEFT or MIDDLE as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing <middle-blank> (the same key that retrieves the previous buffer when issued at the end of a line).

20.7.3 Display Editing Commands

On edit-key terminals (Datamedia): In Interlisp-10, TTYIN reads from the terminal in binary mode, allowing many more editing commands via the edit key, in the style of TVEDIT commands. Note that due to Tenex's unfortunate way of handling typeahead, it is not possible to type ahead edit commands before TTYIN has started (i.e., before its prompt appears), because the edit bit will be thrown away. Also, since ESCAPE has numerous other meanings in Lisp and even in TTYIN (for completion), ESCAPE is not used as a substitute for the edit key.

In Interlisp-D: Users will probably have little use for most of these commands, as cursor positioning can often be done more conveniently, and certainly more obviously, with the mouse. Nevertheless, some commands, such as the case changing commands, can be useful. The <bottom-blank> key can be used as an edit (meta) key in Chorus and subsequent releases if you perform (TTYINMETA T). This calls (METASHIFT T) to enable the meta key, redefines the middle and top blank keys, and informs TTYIN that you want to use them. Alternatively, you can use the EDITPREFIXCHAR (by default on <top-blank>) as described in the next paragraph.

On edit-keyless display terminals (Heath): If you want to type any of these commands, you need to prefix them with the "edit prefix" character. Set the variable EDITPREFIXCHAR to the character code of the desired prefix char. Type the edit prefix twice to give an "edit-ESCAPE" command. Some users of the TENEX TVEDIT program like to make ESCAPE (33Q) be the edit prefix, but this makes it somewhat awkward to ever use escape completion.

On edit-keyless hardcopy terminals: You probably want to ignore this section, since you won't be able to see what's going on when you issure edit commands; there is no attempt made to echo anything reasonable.

In the descriptions below, "current word" means the word the cursor is under, or if under a space, the previous word. Currently parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands. The notation [*CHAR*] means edit-*CHAR*, if you have an edit key, or <editprefixchar> *CHAR* if you don't; $ = escape. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the edit key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the arg. Most of these commands are taken from the display editors TVEDIT and/or E, and are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands:

[delete], [bs], [<]
Back up one (or n) characters.

[space], [>]
Move forward one (or n) characters.

[↑] Moves up one (or n) lines.

[lf] Moves down one (or n) lines.

[(] Move back one (or n) words.

[)] Move ahead one (or n) words.

[tab] Moves to end of line; with an argument moves to nth end of line; [$tab] goes to end of buffer.

[control-L]
Moves to start of line (or nth previous, or start of buffer).

[{] and [}]
Go to start and end of buffer, respectively (like [$control-L] and [$tab]).

[[ ] (edit-left-bracket)
Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Flags".)

[ ]] (edit-right-bracket)
Moves to end of current list.

[Sx] Skips ahead to next (or nth) occurrence of character x, or rings the bell.

[Bx] Backward search, i.e., short for [-S] or [-nS].

Buffer Modification Commands:

[Zx] Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command yet.

[A] or [R]
Repeat the last S, B or Z command, regardless of any intervening input (note this differs from Tvedit's A command).

[K] Kills the character under the cursor, or n chars starting at the cursor.

[I] Begin inserting. Exit insert with any edit command. Characters you type will be inserted, rather than overwriting the existing text. If EMACSFLG (page 20.43) is true (default in Interlisp-D), you are always in insert mode, and this command is a noop.

Inserting <cr> behaves slightly different from in tvedit. The sequence [I<cr>] behaves as in TVEDIT; it inserts a blank line ahead of the cursor. <cr> typed any other time while in insert mode actually inserts a <cr>, behaving somewhat like TVEDIT's [B]. [$I] is the same as [I<cr>].

[cr] When the buffer is empty is the same as <lf>, i.e. restores buffer's previous contents. Otherwise is just like a <cr> (except that it also terminates an insert). Thus, [<cr><cr>] will repeat the previous input (as will <lf><cr> without the edit key).

[O] Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

[T] Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle funny cases, such as tabs.

[G] Grabs the contents of the previous line from the cursor position onward. [nG] grabs the nth previous line.

[L] Lowercases current word, or n words on line. [$L] lowercases the rest of the line, or if given at the end of line lowercases the entire line.

[U] Uppercases analogously.

[C] Capitalize. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

[control-Q]
Deletes the current line. [$control-Q] deletes from the current cursor position to the end of the buffer. No other arguments are handled.

[control-W]
Deletes the current word, or the previous word if sitting on a space.

[D<del>] and [D<cr>]
Are the same as [control-W] and [control-Q], for approximate compatibility with TVEDIT.

[J] "Justify" this line. This will break it if it is too long, or move words up from the next line if too short. Will not join to an empty line, or one starting with a tab (both of which are interpreted as paragraph breaks). Any new line breaks it introduces are considered spaces, not carriage returns. [nJ] justifies n lines.

The linelength is defined as TTYJUSTLENGTH, ignoring any prompt characters at the margin. If TTYJUSTLENGTH is negative, it is interpreted as relative to the right margin. TTYJUSTLENGTH is initially −8 in Interlisp-D, 72 in Interlisp-10.

[$F] "Finishes" the input, regardless of where the cursor is. Specifically, it goes to the end of the input and enters a <cr>, control-Z or "]", depending on whether normal, REPEAT or READ input is happening. Note that a "]" won't necessarily end a READ, but it seems likely to in most cases where you would be inclined to use this command, and makes for more predictable behavior.

Miscellaneous Commands:

[P] Interlisp-D: Prettyprint buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

[N] Refresh line. Same as control-R. [$N] refreshes the whole buffer; [nN] refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the screen; if you do a control-T, or a system message appears, or line noise occurs, you may need to refresh the line for best results. In Interlisp-10, if for some reason your terminal falls out of binary mode (e.g. can happen when returning to a Lisp running in a lower fork), Edit-<anything> is unreadable, so you'd have to type control-R instead.

[control-Y]
Gets userexec. Thus, this is like regular control-Y, except when doing a READ (when control-Y is a read macro and hence does not invoke this function).

[$control-Y]
Gets a userexec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for the Lisp executive, you can do [control-L$control-Y] and give it to Lisp.

[←] Adds the current word to the spelling list USERWORDS. With zero arg, removes word. See TTYINCOMPLETEFLG (page 20.44).

Note to Datamedia, Heath users: In addition to simple cursor movement commands and insert/delete, TTYIN uses the display's cursor-addressing capability to optimize cursor movements longer than a few characters, e.g. [tab] to go to the end of the line. In order to be able to address the cursor, TTYIN has to know where it is to begin with. Lisp keeps track of the current print position within the line, but does not keep track of the line on the screen (in fact, it knows precious little about displays, much like Tenex). Thus, TTYIN establishes where it is by forcing the cursor to appear on the last line of the screen. Ordinarily this is the case anyway (except possibly on startup), but if the cursor happens to be only halfway down the screen at the time, there is a possibly unsettling leap of the cursor when TTYIN starts.

20.7.4 Using TTYIN for Lisp Input

When TTYIN is loaded, or a sysout containing TTYIN is started up, the function SETREADFN is called. If the terminal is a display, it sets LISPXREADFN to be TTYINREAD; if the terminal is non-display, SETREADFN will set the variable back to READ. (SETREADFN 'READ) will also set it back to READ.

There are two principal differences between TTYINREAD and READ: (1) parenthesis balancing. The input does not activate on an exactly balancing right paren/bracket unless the input started with a paren/bracket, e.g., "USE (FOO) FOR (FIE)" will all be on one line, terminated by <cr>; and (2) read macros.

In Interlisp-10, TTYIN does not use a read table (TTYIN behaves as though using the default initial Lisp terminal input readtable), so read macros and redefinition of syntax characters are not supported; however, " ' " (QUOTE) and "control-Y" (EVAL) are built in, and a simple implementation of ? and ?= is supplied. Also, the TTYINREADMACROS facility described below can supply some of the functionality of immediate read macros in the editor.

In Interlisp-D, read macros are (mostly) supported. Immediate read macros take effect only if typed at the end of the input (it's not clear what their semantics should be elsewhere).

20.7.5 Useful Macros

There are two useful edit macros that allow you to use TTYIN as a character editor: (1) ED loads the current expression into the ttyin buffer to be edited (this is good for editing comments and strings). Input is terminated in the usual way (by typing a balancing right parenthesis at the end of the input, typing <cr> at the end of an already balanced expression, or control-X anywhere inside the balanced expression). Typing control-E or clearing the buffer aborts ED. (2) EE is like ED but prettyprints the expression into the buffer, and uses its own window. The variable TTYINEDITPROMPT controls what prompt, if any, EE uses; see prompt argument description in next section (the initial setting is no prompt). EE is not yet implemented in Interlisp-10.

The macro BUF loads the current expression into the buffer, preceded by E, to be used as input however desired; as a trivial example, to evaluate the current expression, BUF followed by a <cr> to activate the buffer will perform roughly what the edit macro EVAL does. Of course, you can edit the E to something else to make it an edit command.

BUF is also defined at the executive level as a programmer's assistant command that loads the buffer with the VALUEOF the indicated event, to be edited as desired.

TV is a programmer's assistant command like EV [EDITV] that performs an ED on the value of the variable.

And finally, if the event is considered "short" enough, the programmer's assistant command FIX will load the buffer with the event's input, rather than calling the editor. If you really wanted the Interlisp editor for your fix, you could either say FIX *EVENT* - TTY:, or type control-U (or whatever on tops20) once you got TTYIN's version to force you into the editor.

20.7.6 Programming With TTYIN

(TTYIN *PROMPT SPLST HELP OPTIONS ECHOTOFILE TABS UNREADBUF RDTBL*) [Function]

TTYIN prints *PROMPT*, then waits for input. The value returned in the normal case is a list of all atoms on the line, with comma and parens returned as individual atoms; *OPTIONS* may be used to get a different kind of value back.

PROMPT is an atom or string (anything else is converted to a string). If NIL, the value of DEFAULTPROMPT, initially "** ", will be used. If *PROMPT* is T, no prompt will be given. *PROMPT* may also be a dotted pair ($PROMPT_1$. $PROMPT_2$), giving the prompt for the first and subsequent (or overflow) lines, each prompt being a string/atom or NIL to denote absence of prompt. Note that rebinding DEFAULTPROMPT gives a convenient way to affect all the "ordinary" prompts in some program module.

SPLST is a spelling list, i.e., a list of atoms or dotted pairs (*SYNONYM* . *ROOT*). If supplied, it is used to check and correct user responses, and to provide completion if the user types ESCAPE. If *SPLST* is one of the Lisp system spelling lists (e.g., USERWORDS or SPELLINGS3), words that are escape-completed get moved to the front, just as if a FIXSPELL had found them. Autocompletion is also performed when user types a break character (cr, space, paren, etc), unless one of the "nofixspell" options below is selected; i.e., if the word just typed would uniquely complete by ESCAPE, TTYIN behaves as though ESCAPE had been typed.

HELP, if non-NIL, determines what happens when the user types ? or HELP. If *HELP* = T, program prints back *SPLST* in suitable form. If *HELP* is any other atom, or a string containing no spaces, it performs (DISPLAYHELP *HELP*). Anything else is printed as is. If *HELP* is NIL, ? and HELP are treated as any other atoms the user types. [DISPLAYHELP is a user-supplied function, initially a noop; systems with a suitable HASH package, for example, have defined it to display a piece of text from a hashfile associated with the key HELP.]

OPTIONS is an atom or list of atoms chosen from among the following:

NOFIXSPELL Uses *SPLST* for HELP and Escape completion, but does not attempt any FIXSPELLing. Mainly useful if *SPLST* is incomplete and the caller wants to handle corrections in a more flexible way than a straight FIXSPELL.

MUSTAPPROVE Does spelling correction, but requires confirmation.

CRCOMPLETE Requires confirmation on spelling correction, but also does autocompletion on <cr> (i.e. if what user has typed so far uniquely identifies a member of *SPLST*, completes it). This allows you to have the benefits of autocompletion and still allow new words to be typed.

DIRECTORY (only if *SPLST*=NIL) Interprets Escape to mean directory name completion [Interlisp-10 only].

USER Like DIRECTORY, but does username completion. This is identical to DIRECTORY under Tenex [Interlisp-10 only].

FILE (only if *SPLST*=NIL) Interprets Escape to mean filename completion, i.e. does a GTJFN [Sumex and Tops20 only].

FIX If response is not on, or does not correct to, *SPLST*, interacts with user until an acceptable response is entered. A blank line (returning NIL) is always accepted. Note that if you are willing to accept responses that are not on *SPLST*, you probably should specify one of the options NOXFISPELL, MUSTAPPROVE or CRCOMPLETE, lest the user's new response get FIXSPELLed away without their approval.

STRING — Line is read as a string, rather than list of atoms. Good for free text.

NORAISE — Does not convert lower case letters to upper case.

NOVALUE — For use principally with the *ECHOTOFILE* arg (below). Does not compute a value, but returns T if user typed anything, NIL if just a blank line.

REPEAT — For multi-line input. Repeatedly prompts until user types control-Z (as in Tenex sndmsg). Returns one long list; with STRING option returns a single string of everything typed, with carriage returns (EOL) included in the string.

TEXT — Implies REPEAT, NORAISE, and NOVALUE. Additionally, input may be terminated with control-V, in which case the global flag CTRLVFLG will be set true (it is set to NIL on any other termination). This flag may be utilized in any way the caller desires.

COMMAND — Only the first word on the line is treated as belonging to *SPLST*, the remainder of the line being arbitrary text; i.e., "command format". If other options are supplied, COMMAND still applies to the first word typed. Basically, it always returns (*CMD* . *REST-OF-INPUT*), where *REST-OF-INPUT* is whatever the other options dictate for the remainder. E.g. COMMAND NOVALUE returns (*CMD*) or (*CMD* . T), depending on whether there was further input; COMMAND STRING returns (*CMD* . "*REST-OF-INPUT*"). When used with REPEAT, COMMAND is only in effect for the first line typed; furthermore, if the first line consists solely of a command, the REPEAT is ignored, i.e., the entire input is taken to be just the command.

READ — Parens, brackets, and quotes are treated a la READ, rather than being returned as individual atoms. Control characters may be input via the control-Vx notation. Input is terminated roughly along the lines of READ conventions: a balancing or over-balancing right paren/bracket will activate the input, or <cr> when no parenthesis remains unbalanced. READ overrides all other options (except NORAISE).

LISPXREAD — Like READ, but implies that TTYIN should behave even more like READ, i.e., do NORAISE, not be errorset-protected, etc.

NOPROMPT — Interlisp-D only: The prompt argument is treated as usual, except that TTYIN assumes that the prompt for the first line has already been printed by the caller; the prompt for the first line is thus used only when redisplaying the line.

ECHOTOFILE if specified, user's input is copied to this file, i.e., TTYIN can be used as a simple text-to-file routine if NOVALUE is used. If *ECHOTOFILE* is a list, copies to all files in the list. *PROMPT* is not included on the file.

TABS is a special addition for tabular input. It is a list of tabstops (numbers). When user types a tab, TTYIN automatically spaces over to the next tabstop (thus the first tabstop is actually the second "column" of input). Also treats specially the characters * and “; they echo normally, and then automatically tab over.

UNREADBUF allows the caller to "preload" the TTYIN buffer with a line of input. *UNREADBUF* is a list, the elements of which are unread into the buffer (i.e., "the outer parentheses are stripped off") to be edited further as desired; a simple <cr> (or control-Z for REPEAT input) will thus cause the buffer's contents to be returned unchanged. If doing READ input, the "PRIN2 names" of the input list are used, i.e., quotes and %'s will appear as needed; otherwise the buffer will look as though *UNREADBUF* had been PRIN1'ed. *UNREADBUF* is treated somewhat like READBUF, so that if it contains a pseudo-carriage return (the value of HISTSTR0), the input line terminates there.

Input can also be unread from a file, using the HISTSTR1 format: *UNREADBUF* = (<*value of* HISTSTR1> (*FILE START* . *END*)), where *START* and *END* are file byte pointers. This makes TTYIN a miniature text file editor.

RDTBL [Interlisp-D only] is the read table to use for READing the input when one of the READ options is given. A lot of character interpretations are hardwired into TTYIN, so currently the only effect this has is in the actual READ, and in deciding whether a character typed at the end of the input is an immediate read macro, for purposes of termination.

If the global variable TYPEAHEADFLG is T, or option LISPXREAD is given, TTYIN permits type-ahead; otherwise it clears the buffer before prompting the user.

20.7.7 EE Interface

The following may be useful as a way of outsiders to call TTYIN as an editor. These functions are currently only in Interlisp-D.

(TTYINEDIT *EXPRS WINDOW PRINTFN*) [Function]

This is the body of EE. Switches the tty to *WINDOW*, clears it, prettyprints *EXPRS*, a list of expressions, into it, and leaves you in TTYIN to edit it as Lisp input. Returns a new list of expressions.

If *PRINTFN* is non-NIL, it is a function of two arguments, *EXPRS* and *FILE*, which is called instead of PRETTYPRINT to print the expressions to the window (actually a scratch file). Note that *EXPRS* is a list, so normally the outer parentheses should not be printed. *PRINTFN*=T is shorthand for "unpretty"; use PRIN2 instead of PRETTYPRINT.

TTYINAUTOCLOSEFLG [Variable]

If TTYINAUTOCLOSEFLG is true, TTYINEDIT closes the window on exit.

TTYINEDITWINDOW [Variable]

If the *WINDOW* arg to TTYINEDIT is NIL, it uses the value of TTYINEDITWINDOW, creating it if it does not yet exist.

TTYINPRINTFN [Variable]

The default value for *PRINTFN* in EE's call to TTYINEDIT.

(SET.TTYINEDIT.WINDOW *WINDOW*) [Function]

Called under a RESETLST. Switches the tty to *WINDOW* (defaulted as in TTYINEDIT) and clears it. The window's position is left so that TTYIN will be happy with it if you now call TTYIN yourself. Specifically, this means positioning an integral number of lines from the bottom of the window, the way the top-level tty window normally is.

`(TTYIN.SCRATCHFILE)` [Function]

Returns, possibly creating, the scratchfile that TTYIN uses for prettyprinting its input. The file pointer is set to zero. Since TTYIN does use this file, beware of multiple simultaneous use of the file.

20.7.8 ?= Handler

In Interlisp, the ?= read macro displays the arguments to the function currently "in progress" in the typein. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it processes this macro specially on its own, printing the arguments below your typein and then putting the cursor back where it was when ?= was typed. For users who want special treatment of ?=, the following hook exists:

`TTYIN?=FN` [Variable]

The value of this variable, if non-`NIL`, is a user function of one argument that is called when ?= is typed. The argument is the function that ?= thinks it is inside of. The user function should return one of the following:

`NIL` Normal ?= processing is performed.

`T` Nothing is done. Presumably the user function has done something privately, perhaps diddled some other window, or called `TTYIN.PRINTARGS` (below).

a list `(ARGS . `*`STUFF`*`)`
Treats *STUFF* as the argument list of the function in question, and performs the normal ?= processing using it.

anything else
The value is printed in lieu of what ?= normally prints.

At the time that ?= is typed, nothing has been "read" yet, so you don't have the normal context you might expect inside a conventional readmacro. If the user function wants to examine the typed-in arguments being passed to the fn, however, it can perform `(TTYIN.READ?=ARGS)`, which bundles up everything between the function and the typing of ?= into a list, which it returns (thus it parallels an arglist; `NIL` if ?= was typed immediately after the function name).

`(TTYIN.PRINTARGS` *FN ARGS ACTUALS ARGTYPE*`)` [Function]

Does the function/argument printing for ?=. *ARGS* is an argument list, *ACTUALS* is a list of actual parameters (from the typein) to match up with args. *ARGTYPE* is a value of the function `ARGTYPE`; it defaults to `(ARGTYPE` *FN*`)`.

20.7.9 Read Macros

When doing `READ` input in Interlisp-10, no Lisp-style read macros are available (but the ' and control-Y macros are built in). Principally because of the usefulness of the editor read macros (set by

SETTERMCHARS), and the desire for a way of changing the meanings of the display editing commands, the following exists as a hack:

TTYINREADMACROS [Variable]

Value is a set of shorthand inputs useable during READ input. It is an alist of entries (*CHARCODE* . *SYNONYM*). If the user types the indicated character (edit bit is denoted by the 200Q bit in charcode), TTYIN behaves as though the synonym character had been typed.

Special cases: 0 - the character is ignored; 200Q - pure Edit bit; means to read another char and turn on its edit bit; 400Q - macro quote: read another char and use its original meaning. For example, if you have macros ((33Q . 200Q) (30Q . 33Q)), then Escape (33Q) will behave as an edit prefix, and control-X (30Q) will behave like Escape. Note: currently, synonyms for edit commands are not well-supported, working only when the command is typed with no argument.

Slightly more powerful macros also can be supplied; they are recognized when a character is typed on an empty line, i.e., as the first thing after the prompt. In this case, the TTYINREADMACROS entry is of the form (*CHARCODE* T . *RESPONSE*) or (*CHARCODE* *CONDITION* . *RESPONSE*), where *CONDITION* is a list that evaluates true. If *RESPONSE* is a list, it is EVALed; otherwise it is left unevaluated. The result of this evaluation (or *RESPONSE* itself) is treated as follows:

NIL The macro is ignored and the character reads normally, i.e., as though TTYINREADMACROS had never existed.

An integer

A character code, treated as above. Special case: -1 is treated like 0, but says that the display may have been altered in the evaluation of the macro, so TTYIN should reset itself appropriately.

Anything else

This TTYIN input is terminated (with a crlf) and returns the value of "response" (turned into a list if necessary). This is the principal use of this facility. The macro character thus stands for the (possibly computed) reponse, terminated if necessary with a crlf. The original character is not echoed.

Interrupt characters, of course, cannot be read macros, as TTYIN never sees them, but any other characters, even non-control chars, are allowed. The ability to return NIL allows you to have conditional macros that only apply in specified situations (e.g., the macro might check the prompt (LISPXID) or other contextual variables). To use this specifically to do immediate editor read macros, do the following for each edit command and character you want to invoke it with:

```
(ADDTOVAR TTYINREADMACROS (CHARCODE 'CHARMACRO? EDITCOM))
```

For example, (ADDTOVAR TTYINREADMACROS (12Q CHARMACRO? !NX)) will make linefeed do the !NX command. Note that this will only activate linefeed at the beginning of a line, not anywhere in the line. There will probably be a user function to do this in the next release.

Note that putting (12Q T . !NX) on TTYINREADMACROS would also have the effect of returning "!NX" from the READ call so that the editor would do an !NX. However, TTYIN would also return !NX outside the editor (probably resulting in a u.b.a. error, or convincing DWIM to enter the editor), and also the clearing of the output buffer (performed by CHARMACRO?) would not happen.

20.7.10 Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. Their initial values are all NIL. In Interlisp-D, the flags are all initially T.

TYPEAHEADFLG [Variable]

If true, TTYIN always permits typeahead; otherwise it clears the buffer for any but LISPXREAD input.

?ACTIVATEFLG [Variable]

If true, enables the feature whereby ? lists alternative completions from the current spelling list.

EMACSFLG [Variable]

Affects display editing. When true, TTYIN tries to behave a little more like EMACS (in very simple ways) than TVEDIT. Specifically, it has the following effects currently: (1) all non-edit characters self-insert (i.e. behave as if you're always in Insert mode); (2) [D] is the EMACS delete to end of word command.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis/bracket, TTYIN will briefly move the cursor to the matching parenthesis/bracket, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you'll never notice it). This feature was inspired by a similar EMACS feature, and turned out to be pretty easy to implement.

TTYINBSFLG [Variable]

Causes TTYIN to always physically backspace, even if you're running on a non-display (not a DM or Heath), rather than print \deletedtext\ (this assumes your hardcopy terminal or glass tty is capable of backspacing). If TTYINBSFLG is LF, then in addition to backspacing, TTYIN x's out the deleted characters as it backs up, and when you stop deleting, it outputs a linefeed to drop to a new, clean line before resuming. To save paper, this linefeed operation is not done when only a single character is deleted, on the grounds that you can probably figure out what you typed anyway.

TTYINRESPONSES [Variable]

An alist of special responses that will be handled by routines designated by the programmer. See "Special Responses", below.

TTYINERRORSETFLG [Variable]

[Interlisp-D only] If true, non-LISPXREAD inputs are errorset-protected (control-E traps back to the prompt), otherwise errors propagate upwards. Initially NIL.

TTYINMAILFLG [Variable]

[Tenex only] When true, performs mail checking, etc. before most inputs (except EVALQT inputs, where it is assumed this has already been done, or inputs indented by more than a few spaces). The MAILWATCH package must be loaded for this.

TTYINCOMPLETEFLG [Variable]

If true, enables Escape completion from USERWORDS during READ inputs. Details below.

USERWORDS (page 15.15) contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing "EF xx$") or type a call to it. If there is no completion for the current word from USERWORDS, the escape echoes as "$", i.e. nothing special happens; if there is more than one possible completion, you get beeped. If typed when not inside a word, Escape completes to the value of LASTWORD, i.e., the last thing you typed that the p.a. "noticed" (setting TTYINCOMPLETEFLG to 0 disables this latter feature), except that Escape at the beginning of the line is left alone (it is a p.a. command).

If you really wanted to enter an escape, you can, of course, just quote it with a control-V, like you can other control chars.

You may explicitly add words to USERWORDS yourself that wouldn't get there otherwise. To make this convenient online the edit command [←] means "add the current atom to USERWORDS" (you might think of the command as "pointing out this atom"). For example, you might be entering a function definition and want to "point to" one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from USERWORDS.

Note that this feature loses some of its value if the spelling list is too long, for then the completion takes too long computationally and, more important, there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list USERWORDS keeps the "temporary" section (which is where everything goes initially unless you say otherwise) limited to #USERWORDS atoms, initially 100. Words fall off the end if they haven't been used (they are "used" if FIXSPELL corrects to one, or you use <escape> to complete one).

20.7.11 Special Responses

There is a facility for handling "special responses" during any non-READ TTYIN input. This action is independent of the particular call to TTYIN, and exists to allow you to effectively "advise" TTYIN to intercept certain commands. After the command is processed, control returns to the original TTYIN call. The facility is implemented via the list TTYINRESPONSES.

TTYINRESPONSES [Variable]

TTYINRESPONSES is a list of elements, each of the form:

(*COMMANDS RESPONSE-FORM OPTION*)

COMMANDS is a single atom or list of commands to be recognized; *RESPONSE-FORM* is EVALed (if a list), or APPLYed (if an atom) to the command and the rest of the line. Within this form one can reference the free variables COMMAND (the command the user typed) and LINE (the rest of the line). If *OPTION* is the atom LINE, this means to pass the rest of line as a list; if it is STRING, this means to pass it as a string; otherwise, the command is only valid if there is nothing else on the line. If *RESPONSE-FORM* returns the atom IGNORE, it is not treated as a special response (i.e. the input is returned normally as the result of TTYIN).

In MYCIN, the COMMENT command is handled this way; any time the user types COMMENT as the first word of input, TTYIN passes the rest of the line to a mycin-defined function which prompts for the text of the comment (recursively using TTYIN with the TEXT option). When control returns, TTYIN goes back and prompts for the original input again. The TTYINRESPONSES entry for this is (COMMENT (GRIPE LINE) LIST); GRIPE is a MYCIN function of one argument (the one-line comment, or NIL for extended comments).

Suggested use: global commands or options can be added to the toplevel value of TTYINRESPONSES. For more specialized commands, rebind TTYINRESPONSES to (APPEND *NEWENTRIES* TTYINRESPONSES) inside any module where you want to do this sort of special processing.

Special responses are not checked for during READ-style input.

20.7.12 Display Types

[This is not relevant in Interlisp-D]

TTYIN determines the type of display by calling DISPLAYTERMP, which is initially defined to test the value of the GTTYP jsys. It returns either NIL (for printing terminals) or a small number giving TTYIN's internal code for the terminal type. The types TTYIN currently knows about:

0 = glass tty (capable of deleting chars by backspacing, but little else);

1 = Datamedia;

2 = Heath.

Only the Datamedia has full editing power. DISPLAYTERMP has built into it the correct terminal types for Sumex and Stanford campus 20's: Datamedia = 11 on tenex, 5 on tops20; Heath = 18 on Tenex, 25 on tops20. You can override those values by setting the variable DISPLAYTYPES to be an alist associating the GTTYP value with one of these internal codes. For example, Sumex displays correspond to DISPLAYTYPES = ((11 . 1) (18 . 2)) [although this is actually compiled into DISPLAYTERMP for speed]. Any display terminal other than Datamedia and Heath can probably safely be assigned to "0" for glass tty.

To add new terminal types, you have to choose a number for it, add new code to TTYIN for it and recompile. The TTYIN code specifies what the capabilities of the terminal are, and how to do the primitive operations: up, down, left, right, address cursor, erase screen, erase to end of line, insert character, etc.

For terminals lacking an Edit key (currently only Datamedias have it), set the variable EDITPREFIXCHAR to the ascii code of an Edit "prefix" (i.e. anything typed preceded by the prefix is considered to have the edit bit on). If your EDITPREFIXCHAR is 33Q (Escape), you can type a real Escape by typing 3 of them (2 won't do, since that means "Edit-Escape", a legitimate argument to another command). You could also define an Escape synonym with TTYINREADMACROS if you wanted (but currently it doesn't work in filename completion). Setting EDITPREFIXCHAR for a terminal that is not equipped to handle the full range of editing functions (only the Heath and Datamedia are currently so equipped) is not guaranteed to work, i.e. the display will not always be up to date; but if you can keep track of what you're doing, together with an occasional control-R to help out, go right ahead.

CHAPTER 21

ETHERNET

Interlisp was first developed on large timesharing machines which provided each user with access to large amounts of disk storage, printers, mail systems, etc. Interlisp-D, however, was designed to run on smaller, single-user machines without these facilities. In order to provide Interlisp-D users with access to all of these services, Interlisp-D supports the Ethernet communications network, which allows multiple Interlisp-D machines to share common printers, file servers, etc.

Interlisp-D supports the Experimental Ethernet (3 Megabits per second) and the Ethernet (10 Megabits per second) local communications networks. These networks may be used for accessing file servers, remote printers, mail servers, or other machines. This chapter is divided into three sections: First, an overview of the various Ethernet and Experimental Ethernet protocols is presented. Then follow sections documenting the functions used for implementing PUP and NS protocols at various levels.

21.1 ETHERNET PROTOCOLS

The members of the Xerox 1100 family (1100, 1108, 1132), Xerox file servers and laser xerographic printers, along with machines made by other manufacturers (most notably DEC) have the capability of communicating over 3 Megabit per second Experimental Ethernets, 10 Megabit per second Ethernets and telephone lines.

Xerox pioneered its work with Ethernet using a set of protocols known as PARC Universal Packet (PUP) computer communication protocols. The architecture has evolved into the newer Network Systems (NS) protocols developed for use in Xerox office products. All of the members of the Xerox 1100 family can use both NS and PUP protocols.

21.1.1 Protocol Layering

The communication protocols used by the members of the Xerox 1100 family are implemented in a "layered" fashion, which means that different levels of communication are implemented as different protocol layers. Protocol Layering allows implementations of specific layers to be changed without requiring changes to any other layers. The layering also allows use of the same higher level software with different lower levels of protocols. Protocol designers can implement new types of protocols at the correct protocol level for their specific application in a layered system.

At the bottom level, level zero, there is a need to physically transmit data from one point to another. This level is highly dependent on the particular transmission medium involved. There are many different level zero protocols, and some of them may contain several internal levels. At level one, there is a need to decide where the data should go. This level is concerned with how to address a source and destination, and how to choose the correct transmission medium to use in order to route the packet towards its destination. A level one packet is transmitted by *encapsulating* it in the level zero packet appropriate for the transmission medium selected. For each independent communication protocol system, a single level one protocol is defined. The rule for delivery of a level one packet is that the communication system must only make a best effort to deliver the packet. There is no guarantee that the packet is delivered, that the packet is not duplicated and delivered twice, or that the packets will be delivered in the same order as they were sent.

The addresses used in level zero and level one packets are not necessarily the same. Level zero packets are specific to a particular transmission medium. For example, the destination address of a level zero packet transmitted on one of the two kinds of Ethernet is the Ethernet address (host number) of a machine on the particular network. Level one packets specify addresses meaningful to the particular class of protocols being implemented. For the PUP and NS protocols, the destination address comprises a network number, host number (not necessarily the same as the level zero host number), and a socket number. The socket number is a higher-level protocol concept, used to multiplex packets arriving at a single machine destined for separate logical processes on the machine.

Protocols in level two add order and reliability to the level one facilities. They suppress duplicate packets, and are responsible for retransmission of packets for which acknowledgement has not been received. The protocol layers above level two add conventions for data structuring, and implement application specific protocols.

21.1.2 Level Zero Protocols

Level zero protocols are used to physically connect computers. The addresses used in level zero protocols are protocol specific. The Ethernet and Experimental Ethernet level zero protocols use host numbers, but level zero phone line protocols contain less addressing information since there are only two hosts connected to the telephone line, one at each end. As noted above, a level zero protocol does not include network numbers.

The 3MB Experimental Ethernet [1] was developed at PARC. Each Experimental Ethernet packet includes a source and destination host address of eight bits. The Experimental Ethernet standard is used by any machine attached to an Experimental Ethernet.

The 10MB Ethernet [2] was jointly developed and standardized by Digital, Intel, and Xerox. Each Ethernet level zero packet includes a source and destination host address that is 48 bits long. The Ethernet standard is used by any machine attached to an Ethernet.

Both of the level one protocols described later (PUP and NS) can be transported on any of the level zero protocols described above.

The Ethernet and Experimental Ethernet protocols are broadcast mediums. Data packets can be sent on these networks to every host attached to the net. A packet directed at every host on a network is a broadcast packet.

Other Level 0 protocols in use in industry include X.25, broadband networks, and Chaosnet. In addition, by using the notion of "mutual encapsulation", it is possible to treat a higher-level protocol (e.g. ARPANET) as if it were a Level Zero Protocol.

21.1.3 Level One Protocols

Two Level One Protocols are used in the Xerox 1100 Family, the PUP and the NS protocols. With the proper software, computers attached to Ethernets or Experimental Ethernets can send PUPs and NS packets to other computers on the same network, and to computers attached to other Ethernets or Experimental Ethernets.

The PUP protocols [3] were designed by Xerox computer scientists at the Palo Alto Research Center. The destination and source addresses in a PUP packet are specified using an 8-bit network number, an 8-bit host number, and a 32-bit socket number. The 8-bit network number allows an absolute maximum of 256 PUP networks in an internet. The 8-bit host number is network relative. That is, there may be many host number "1"s, but only one per network. 8 bits for the host number limits the number of hosts per network to 256. The socket number is used for further levels of addressing within a specific machine.

The Network Systems (NS) protocols [4, 5] were developed by the Xerox Office Products Division. Each NS packet address includes a 32-bit network number, a 48-bit host number, and a 16-bit socket number. The NS host and network numbers are unique through all space and time. A specific NS host number is generally assigned to a machine when it is manufactured, and is never changed. In the same fashion, all networks (including those sold by Xerox and those used within Xerox) use the same network numbering space—there is only one network "74".

21.1.4 Higher Level Protocols

The higher level PUP protocols include the File Transfer Protocol (FTP) and the Leaf Protocol used to send and retrieve files from Interim File Servers (IFSs) and DEC File Servers, the Telnet protocol implemented by "Chat" windows and servers, and the EFTP protocol used to communicate with the laser xerographic printers developed by PARC ("Dovers" and "Penguins").

The higher level NS protocols include the Filing Protocol which allows workstations to access the product File Services sold by Xerox, the Clearinghouse Protocol used to access product Clearinghouse Services, and the TelePress Protocol used to communicate with the Xerox model 8044 Print Server.

21.1.5 Connecting Networks: Routers and Gateways

When a level one packet is sent from one machine to another, and the two machines are not on the same network, the packet must be passed between networks. Computers that are connected to two or more level zero mediums are used for this function. In the PUP world, these machines have been historically called "Gateways." In the NS world these machines are called Internetwork Routers (Routers), and the function is packaged and sold by Xerox as the Internetwork Routing Service (IRS).

Every host that uses the PUP protocols requires a PUP address; NS Hosts require NS addresses. An address consists of two parts: the host number and the network number. A computer learns its network number by communicating with a Router or Gateway that is attached to the same network. Host number determination is dependent on the hardware and the type of host number, PUP or NS.

21.1.6 Addressing Conflicts with Level Zero Mediums

For convenience in the respective protocols, a level one PUP (8-bit) host number is the same as a level zero Experimental Ethernet host number; i.e., when a PUP level one packet is transported by an Experimental Ethernet to another host on the same network, the level zero packet specifies the same host number as the level one packet. Similarly, a level one NS (48-bit) host number is the same as a level zero Ethernet host number.

When a PUP level one packet is transported by an Ethernet, or an NS level one packet is sent on Experimental Ethernet, the level one host number cannot be used as the level zero address, but rather some means must be provided to determine the correct level zero address. Xerox solved this problem by specifying another level-one protocol called *translation* to allow hosts on an Experimental Ethernet to announce their NS host numbers, or hosts on an Ethernet to announce their PUP host numbers. Thus, both the Ethernet and Experimental Ethernet Level Zero Protocols totally support both families of higher level protocols.

21.1.7 References

[1] Robert M. Metcalfe and David R. Boggs, Ethernet: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM*, vol. 19 no. 7, July 1976.

[2] Digital Equipment Corporation, Intel Corporation, Xerox Corporation. The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications. September 30, 1980, Version 1.0

[3] D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe, PUP: An Internetwork Architecture, *IEEE Transactions on Communications*, com-28:4, April 1980.

[4] Xerox Corporation. Courier: The Remote Procedure Call Protocol. Xerox System Integration Standard. Stamford, Connecticut, December, 1981, XSIS 038112.

[5] Xerox Corporation. Internet Transport Protocols. Xerox System Integration Standard. Stamford, Connecticut, December, 1981, XSIS 028112.

21.2 HIGHER-LEVEL PUP PROTOCOL FUNCTIONS

This section describes some of the functions provided in Interlisp-D to perform protocols above Level One. Level One functions are described in a later section, for the benefit of those users who wish to program new protocols.

The following functions provide assorted network services.

(ETHERHOSTNUMBER *NAME*) [Function]

Returns the number of the named host. The number is 16-bit quantity, the high 8 bits designating the net and the low 8 bits the host. If *NAME* is NIL, returns the number of the local host.

(ETHERPORT *NAME ERRORFLG MULTFLG*) [Function]

Returns a port corresponding to *NAME*. A "port" is a network address that represents (potentially) one end of a network connection, and includes a socket number in addition to the network and host numbers. Most network functions that take a port as argument allow the socket to be zero, in which case a well-known socket is supplied. A port is currently represented as a dotted pair (*NETHOST* . *SOCKET*).

NAME may be a litatom, in which case its address is looked up, or a port, which is just returned directly. If *ERRORFLG* is true, generates an error "host not found" if the address lookup fails, else it returns NIL. If *MULTFLG* is true, returns a list of alternative port specifications for *NAME*, rather than a single port (this is provided because it is possible for a single name in the name database to have multiple addresses). If *MULTFLG* is NIL and *NAME* has more than one address, the currently nearest one is returned. ETHERPORT caches its results.

The *SOCKET* of a port is usually zero, unless the name explicitly contains a socket designation, a number or symbolic name following a + in *NAME*, e.g., PHYLUM+LEAF. A port can also be specified in the form "net#host#socket", where each of net, host and socket is a sequence of octal digits; the socket, but not the terminating #, can be omitted, in which case the socket is zero.

(ETHERHOSTNAME *PORT USE.OCTAL.DEFAULT*) [Function]

Looks up the name of the host at address *PORT*. *PORT* may be a numeric address, a (*NETHOST* . *SOCKET*) pair returned from ETHERPORT, or a numeric designation in string form, "net#host#socket", as described above. In the first case, the net defaults to the local net. If *PORT* is NIL, returns the name of the local host. If there is no name for the given port, but *USE.OCTAL.DEFAULT* is true, the function returns a string specifying the port in octal digits, in the form "*NET#HOST#SOCKET*", with *SOCKET* omitted if it is zero. Most functions that take a port argument will also accept ports in this octal format.

(PRINTERSTATUS *PRINTERNAME*) [Function]

Returns status of *PRINTERNAME*, the name of a Press Printer, in the form (*CODE* . "readable string"). Returns NIL if the printer does not respond in a reasonable time, which can occur if the printer is very busy, or does not implement the printer status service. *CODE* is interpreted as follows:

1 Printer is not spooling (down for servicing)

2 Printer is idle

3 Printer is busy (printing or accepting a file)

(EFTP *HOST FILE PRINTERFLG #SIDES*) [Function]

Transmits *FILE* to *HOST* using the EFTP protocol. The *FILE* need not be open on entry, but in any case is closed on exit. The principal use of the EFTP protocol is for transmitting Press files to a printer. If *PRINTERFLG* is non-NIL, assumes that *HOST* is a printer and *FILE* is a press file, and takes additional action: it performs a PRINTERSTATUS for *HOST* and prints this information to the prompt window; and it fills in the "printed-by" field on the last page of the press file with USERNAME, and the "copies" field with (OR (FIXP *PRINTERFLG*) 1). For printers capable of duplex printing, *#SIDES* may be 1 or 2, meaning print one- or two-sided, respectively; NIL means use the printer's default. EFTP returns only on success; if *HOST* does not respond, it keeps trying.

21.3 HIGHER-LEVEL NS PROTOCOL FUNCTIONS

The following is a description of the Interlisp-D facilities for using Xerox SPP and Courier protocols and the services based on them.

21.3.1 SPP Stream Interface

This section describes the stream interface to the Sequenced Packet Protocol.

(SPP.OPEN *HOST SOCKET PROBEP NAME*) [Function]

This function is used to open an SPP stream. If *HOST* is specified, an SPP connection is initiated to *HOST* with remote socket *SOCKET*. If both *HOST* and *PROBEP* are specified, then the connection is probed for a response before returning the stream; NIL is returned if *HOST* doesn't respond. If *HOST* is NIL, a passive connection is created which listens for an incoming connection to local socket *SOCKET*. *NAME* is a mnemonic name for the connection process, mainly useful for debugging. The function returns an SPP stream, for which the standard stream operations BIN, BOUT, CLOSEF, and EOFP are defined. In particular, COPYBYTES may be used on SPP streams.

The SPP stream that is returned is open for both input and output, since SPP connections are bidirectional. However, the underlying stream I/O functions use only a single buffer. Some care must therefore be exercised to insure that any buffered output data is forced out before any new data is read, and that all data up to a message boundary has been read before any new data is written. Functions described below are used for this purpose. While these restrictions may seem severe, in practice most use of SPP streams is done by the Courier remote procedure call facility, rather than directly by the programmer. Courier conforms to the model of alternating exchanges of messages quite well.

SPP.USER.TIMEOUT [Variable]

Specifies the time, in milliseconds, to wait before deciding that a host isn't responding.

(SPP.FLUSH *STREAM*) [Function]

This function forces any buffered output data to be transmitted.

(SPP.SENDEOM *STREAM*) [Function]

This function forces out any buffered data and causes an End of Message indication to be sent.

(SPP.CLOSE *STREAM ABORT?*) [Function]

This function closes an SPP stream using the reliable termination protocol. If *ABORT?* is not NIL, the stream is closed even if there is an outstanding bulk data transfer in progress.

(SPP.DSTYPE *STREAM DSTYPE*) [Function]

This function gets or sets the current datastream type. If *DSTYPE* is specified, all subsequent packets that are sent will be of this datastream type, until the next call to SPP.DSTYPE. Since this affects the current partially-filled packet, the stream should probably be flushed (via SPP.FLUSH) before this function is called. If *DSTYPE* is not specified, this function returns the datastream type of the current packet being read.

(SPP.READP *STREAM*) [Function]

This function returns T or NIL depending on whether or not there is data to be read without waiting.

(SPP.EOFP *STREAM*) [Function]

This function returns T or NIL depending on whether or not the connection has been closed.

(SPP.EOMP *STREAM*) [Function]

This function returns T or NIL depending on whether or not an End of Message indication has been reached. This will only be true after the last byte of data in the message has been read.

21.3.2 Courier Remote Procedure Call Protocol

(COURIER.OPEN *HOSTNAME SERVERTYPE NOERRORFLG NAME*) [Function]

This function opens a Courier connection to the specified *HOST* and returns an SPP stream. If *HOST* is a LITATOM, string, or list representation of a Clearinghouse name, *SERVERTYPE* should specify what type of server *HOST* is, so that the name may be looked up in the Clearinghouse database. Currently, *SERVERTYPE* must be one of PRINTSERVER or FILESERVER. Normally, this function will retry the connection \MAXETHERTRIES times before generating an error. If *NOERRORFLG* is specified, NIL will be returned if the connection fails. The Courier connection will be given *NAME*, if specified.

(COURIERPROGRAM *NAME* ···) [NLambda NoSpread Function]

This function is used to define Courier programs. The syntax is

```
(COURIERPROGRAM name (programNumber versionNumber)
   TYPES
   ((typeName typeDefinition)
    ...)
   PROCEDURES
   ((procedureName ARGS (argType ...)
                   RESULTS (resultType ...)
                   ERRORS (errorName ...)
                   procedureNumber)
    ...)
   ERRORS
   ((errorName ARGS (argType ...) errorNumber)
        ...))
)
```

Type definitions are written in the Courier template language, described below. Courier types may either be type names that are defined in the current Courier program, qualified names of the form (otherCourierProgram . typeName), or explicit definitions in the template language.

21.3.2.1 Courier Template Language

This section describes how Courier types are described in Interlisp, and how corresponding values are represented. (See also the Courier protocol definition.)

Predefined types:

BOOLEAN is represented by T and NIL; STRING is represented by strings; CARDINAL, INTEGER, LONGCARDINAL, LONGINTEGER, and UNSPECIFIED are represented by integers.

Constructed types:

```
(ENUMERATION (NAME VALUE) ... (NAME VALUE))
(ARRAY LENGTH TYPE)
(SEQUENCE TYPE)
(RECORD (NAME TYPE) ... (NAME TYPE))
(CHOICE (NAME VALUE TYPE) ... (NAME VALUE TYPE))
```

Representation of constructed types in Lisp:

Objects of Courier type (ENUMERATION (UNKNOWN 0) (RED 1) (BLUE 2)) are represented by the litatoms UNKNOWN, RED, and BLUE.

Objects of Courier type (ARRAY 3 INTEGER) are represented by lists of three integers, such as (10 1 59).

Objects of Courier type (SEQUENCE BOOLEAN) are represented by arbitrary-length lists of T and NIL, such as (NIL T T NIL T).

Objects of Courier type

```
(RECORD (NETWORK LONGCARDINAL)
        (HOST (ARRAY 3 CARDINAL))
        (SOCKET CARDINAL))
```

are represented by lists like ((NETWORK 174) (HOST (100 24 363)) (SOCKET 20)).

Objects of Courier type

```
(CHOICE (STATUS 0 (ENUMERATION (BUSY 0) (COMPLETE 1)))
        (MESSAGE 1 STRING))
```

are represented by lists like (STATUS COMPLETE) or (MESSAGE "Your request has completed.").

(COURIER.CALL *STREAM PROGRAM PROCEDURE* $ARG_1 \cdots ARG_N$ *NOERRORFLG*)

[NoSpread Function]

This function calls the remote procedure *PROCEDURE* of the Courier program *PROGRAM*. *STREAM* is the SPP stream returned by COURIER.OPEN. The arguments should be Lisp values appropriate for the Courier types of the corresponding formal parameters of the procedure (defined under the ARGS property for the procedure). Returns results of the Courier types defined under the RESULTS property. If there is only a single result, it is returned, otherwise a list of results is returned. The *NOERRORFLG* argument controls the treatment of remote errors. If *NOERRORFLG* is NIL, a Lisp error will be generated. If *NOERRORFLG* is T, NIL will be returned as the result of the call. If *NOERRORFLG* is RETURNERRORS, the result of the call will be a list consisting of the atom ERROR followed by the Courier name of the error and any arguments.

Examples:

```
(COURIERPROGRAM EXAMPLEPROGRAM (17 1)
   TYPES
   ((PERSON.NAME (RECORD (FIRST.NAME STRING)
                         (MIDDLE (CHOICE
                                   (NAME 0 STRING)
                                   (INITIAL 1 STRING)))
                         (LAST.NAME STRING)))
    (BIRTHDAY (RECORD (YEAR CARDINAL)
                      (MONTH STRING)
                      (DAY CARDINAL))))
   PROCEDURES
   ((GETBIRTHDAY ARGS (PERSON.NAME)
                 RESULTS (BIRTHDAY)
                 3))
)
```

Defines EXAMPLEPROGRAM to be Courier program number 17, version number 1. The example defines two types, PERSON.NAME and BIRTHDAY, and one procedure, GETBIRTHDAY, whose procedure number is 3. The following code could be used to call the remote GETBIRTHDAY procedure on the host with address HOSTADDRESS.

```
(SETQ STREAM (COURIER.OPEN HOSTADDRESS))
(COURIER.CALL STREAM
              (QUOTE EXAMPLEPROGRAM)
              (QUOTE GETBIRTHDAY)
              (QUOTE ((FIRST.NAME "Eric")
                      (MIDDLE (INITIAL "C"))
                      (LAST.NAME "Cooper"))))
```

COURIER.CALL in this example will return a value such as

```
((YEAR 1959) (MONTH "January") (DAY 10))
```

21.3.2.2 Manipulating Courier Representations

Several Courier programs use values of type (SEQUENCE UNSPECIFIED) to handle user-defined or otherwise extensible object types. Often it is necessary to convert between a list of 16 bit words (the sequence of UNSPECIFIEDs) and a Courier value. The following function should be used for this purpose.

(COURIER.READ.REP *LIST.OF.WORDS PROGRAM TYPE*) [Function]

This function returns the Lisp representation of the Courier object of type *TYPE* defined in the Courier program *PROGRAM* whose underlying Courier representation is *LIST.OF.WORDS*.

21.3.2.3 Using Bulk Data Transfer with Courier

Two Courier types are treated specially when they appear in the argument list of a procedure. They are BULK.DATA.SINK and BULK.DATA.SOURCE. A Courier procedure may have at most one such sink or source parameter. The result of a COURIER.CALL on such a procedure is an SPP stream, open for input or output according to whether the bulk data paramter is a sink or a source. The client uses this stream to receive or send the appropriate bulk data object. If the object consists of bytes, this may be done with the usual stream I/O functions such as COPYBYTES. If the data is a stream of Courier objects, the following function should be used.

(COURIER.READ.BULKDATA *STREAM PROGRAM TYPE*) [Function]

STREAM is the bulk data stream returned from COURIER.CALL. *TYPE* is the type of each Courier object in the stream. *PROGRAM* is the Courier program in which *TYPE* is defined. A list of objects of Courier type *TYPE* will be returned.

The observant reader may wonder what happens if the Courier procedure returns one or more results, in addition to taking a bulk data parameter. If a bulk data stream is returned to the caller, what happens to the results? The answer is that the results are collected when the bulk data stream is closed, after the client has transferred the bulk data. The disposition of these results depends on what actual parameter is supplied for the formal bulk data parameter at the time of the call. If it is NIL, the results, if any, will be ignored. Otherwise, the value is assumed to be a function which to be applied to the results. A FUNARG may be used for full generality.

For example, the Courier procedure to print an Interpress master uses a bulk data source to transfer the master, and also returns a request identifier. The Lisp function which performs the COURIER.CALL passes a functional to be called on this request identifier after the stream is closed and printing begins; this functional in turn spawns a process which monitors the progress of the job.

(COURIERTRACE *FLG REGION*) [Function]

This function controls the tracing of Courier remote procedure calls. It is similar to PUPTRACE and XIPTRACE, but operates at the call/return level rather than the packet level.

21.3.3 NS Printing

This section describes the facilities that are available for printing Interpress masters on NS printservers.

NS.DEFAULT.PRINTER [Variable]

The value of this variable is used whenever no printserver is specified for the functions described below. If its value is a LITATOM, string, or Clearinghouse name, the Clearinghouse is queried to find the address of the printserver with that name. If its value is NIL, it will be set automatically to some printserver in the local Clearinghouse domain. In environments where there is no Clearinghouse, the value of NS.DEFAULT.PRINTER must be an appropriate NSADDRESS record.

(OPEN.NS.PRINTING.STREAM *PRINTER DOCUMENT.NAME DOCUMENT.CREATION.DATE SENDER.NAME RECIPIENT.NAME #COPIES MEDIUM PRIORITY STAPLE? TWO.SIDED? NOWATCHDOG?*) [Function]

This function returns a stream for printing an Interpress master on *PRINTER* or on NS.DEFAULT.PRINTER as mentioned above. The caller should write the Interpress data to the stream and then close it using CLOSEF. Printing begins after the stream is closed.

DOCUMENT.NAME is the document name to appear on the header page (a string).

DOCUMENT.CREATION.DATE is the creation date to appear on the header page (a Lisp integer date). The default value is the time of the call.

SENDER.NAME is the name of the sender to appear on the header page (a string). The default value is the name of the user.

RECIPIENT.NAME is the name of the recipient to appear on the header page (a string). The default value is the name of the user.

#COPIES is the number of copies to be printed. The default value is 1.

MEDIUM is the medium on which the master is to be printed. This must be a Courier value of type MEDIUM, which is a list of the form (PAPER (KNOWN.SIZE NAME)), where NAME is one of the LITATOMs US.LETTER, US.LEGAL, A0 through A10, ISO.B0 through ISO.B10, and JIS.B0 through JIS.B10. The default value is determined by the printer.

PRIORITY is the priority of this print request (LOW, NORMAL, or HIGH). The default value is NORMAL.

STAPLE? is T or NIL depending on whether the document should be stapled. The default value is NIL.

TWO.SIDED? is T or NIL depending on whether the document should be printed on two sides. The default value is NIL.

NOWATCHDOG? is non-NIL if the client does not want a watchdog process to monitor the status of the printing job.

(NSPRINT *PRINTER FILE.NAME DOCUMENT.NAME DOCUMENT.CREATION.DATE SENDER.NAME RECIPIENT.NAME #COPIES MEDIUM PRIORITY STAPLE? TWO.SIDED?*) [Function]

This function prints an Interpress master on *PRINTER* or on NS.DEFAULT.PRINTER as mentioned above. *FILE.NAME* should be the name of an Interpress file to be printed. The remaining arguments are all optional, and are as described for OPEN.NS.PRINTING.STREAM above. *DOCUMENT.NAME* defaults to the full name of the file, and *DOCUMENT.CREATION.DATE* defaults to the creation date of the file.

(NSPRINTER.STATUS *PRINTER*) [Function]

This function returns the Courier value resulting from the GET.PRINTER.STATUS call.

(NSPRINTER.PROPERTIES *PRINTER*) [Function]

This function returns the Courier value resulting from the GET.PRINTER.PROPERTIES call.

21.3.4 Clearinghouse

This section describes functions that may be used to access Clearinghouse servers. Note that these functions are used by the NS printing functions if the printserver is specified by name rather than address.

(START.CLEARINGHOUSE *RESTARTFLG*) [Function]

This function enables Clearinghouse access. It performs an expanding ring broadcast in order to find the first Clearinghouse server. If *RESTARTFLG* is non-NIL, the cache of Clearinghouse information is invalidated and a new broadcast is done. This may be necessary if the local Clearinghouse server goes down.

CH.NET.HINT [Variable]

Hint as to which network the local Clearinghouse server is on, for use by START.CLEARINGHOUSE above. If CH.NET.HINT is bound to a network number, that network will be tried first, followed by the others in the routing table. If the local Clearinghouse server is not on the directly connected network, setting CH.NET.HINT to the proper network number in the local INIT file will speed up START.CLEARINGHOUSE considerably.

(SHOW.CLEARINGHOUSE) [Function]

This function displays the structure of the cached Clearinghouse information in a window. Once created, it will be redisplayed whenever the cache is updated. The structure is shown using GRAPHER".

(SHOW.ENTIRE.CLEARINGHOUSE) [Function]

This function attempts to cache information about all the Clearinghouse domains, so that the Clearinghouse structure window will show the entire database.

CH.DEFAULT.DOMAIN [Variable]

This is a string specifying the default Clearinghouse domain. If it is NIL, it will be set automatically by START.CLEARINGHOUSE. Otherwise, it should be set in an INIT file.

CH.DEFAULT.ORGANIZATION [Variable]

This is a string specifying the default Clearinghouse organization. If it is NIL, it will be set automatically by START.CLEARINGHOUSE. Otherwise, it should be set in an INIT file.

(CH.ORGANIZATIONS *ORGANIZATIONPATTERN*) [Function]

This function returns the list of organization names in the Clearinghouse database matching *ORGANIZATIONPATTERN*. The default pattern is "*", which matches anything.

(CH.DOMAINS *DOMAINPATTERN*) [Function]

This function returns the list of domain names in the Clearinghouse database matching *DOMAINPATTERN*. The default pattern is "*", which matches anything.

(CH.ENUMERATE *OBJECTPATTERN PROPERTY*) [Function]

This function returns the list of object names matching *OBJECTPATTERN* and having the property *PROPERTY*. Currently, *PROPERTY* must be one of USER, PRINTSERVER, FILESERVER, and ALL. For example,

```
(CH.ENUMERATE "*:PARC:Xerox" (QUOTE USER))
```

will return a list of the names of users at Xerox PARC.

(CH.LOOKUP.USER *NAME*) [Function]

This function returns the user information for the first user whose name matches *NAME*.

(LOOKUP.NS.SERVER *NAME TYPE*) [Function]

This function returns the NSADDRESS for the first server whose name matches *NAME* and has the property *TYPE*, which must be PRINTSERVER or FILESERVER.

21.3.5 NS Filing

This section describes functions that may be used to access NS fileservers.

21.3.5.1 Pathnames and NS Fileservers

The NS Filing protocol does not support conventional file system pathnames directly. However, the Interlisp-D software that supports access to NS fileservers uses IFS-style pathnames and does the appropriate mapping in software. One important difference, however, is that fileserver, directory, and file names may have spaces in them, each of which must be preceded by a percent sign. The name of an NS fileserver is required to have a colon in it. Thus, even if the fileserver is in the local Clearinghouse domain, a trailing colon should be appended to the name. Case is not significant. For example,

```
{LISPFILE:}<LISPDRAWER>XYZ;3
```

is a valid name for a file on the NS fileserver "LispFile:Parc Place:Xerox".

(NSDIRECTORY *PATTERN*) [Function]

This function returns a list of file names in *PATTERN*, which must be the NS pathname for a directory. (Any wildcards in the name field of the pathname are ignored.)

(NSCREATEDIRECTORY *HOST/DIR*) [Function]

This function creates a new directory with pathname *HOST/DIR*. Top level directories ("file drawers") cannot be created in this way.

(CLOSE.NSFILING.CONNECTIONS) [Function]

This function closes any open connections to NS fileservers.

21.4 LEVEL ONE ETHER PACKET FORMAT

The datatype ETHERPACKET is the vehicle for all kinds of packets transmitted on an Ethernet or Experimental Ethernet. An ETHERPACKET contains several fields for use by the Ethernet drivers and a large, contiguous data area making up the data of the level zero packet. The first several words of the area are reserved for the level one to zero encapsulation, and the remainder (starting at field EPBODY) make up the level one packet. Typically, each level one protocol defines a BLOCKRECORD that overlays the ETHERPACKET starting at the EPBODY field, describing the format of a packet for that particular protocol. For example, the records PUP and XIP define the format of level one packets in the PUP and NS protocols.

The extra fields in the beginning of an ETHERPACKET have mostly a fixed interpretation over all protocols. Among the interesting ones are:

EPLINK A pointer used to link packets, used by the SYSQUEUE mechanism (page 21.25). Since this field is used by the system for maintaining the free packet queue and ether transmission queues, do not use this field unless you understand it.

EPFLAGS A byte field that can be used for any purpose by the user.

EPUSERFIELD A pointer field that can be used for any purpose by the user. It is set to NIL when a packet is released.

EPTRANSMITTING A flag that is true while the packet is "being transmitted", i.e., from the time that the user instructs the system to transmit the packet until the packet is gathered up from the transmitter's finished queue. While this flag is true, the user must *not* modify the packet.

EPREQUEUE A pointer field that specifies the desired disposition of the packet after transmission. The possible values are: NIL means no special treatment; FREE means the packet is to be released after transmission; an instance of a SYSQUEUE means the packet is to be enqueued on the specified queue (page 21.25).

The normal life of an outgoing Ether packet is that a program obtains a blank packet, fills it in according to protocol, then sends the packet over the Ethernet. If the packet needs to be retained for possible retransmission, the EPREQUEUE field is used to specify a queue to place the packet on after its transmission, or the caller hangs on to the packet explicitly.

There are redefinitions, or "overlays" of the ETHERPACKET record specifically for use with the PUP and NS protocols. The following sections describe those records and the handling of the PUP and NS level one protocols, how to add new level one protocols, and the queueing mechanism associated with the EPREQUEUE field.

21.5 PUP LEVEL ONE FUNCTIONS

The functions in this section are used to implement level two and higher PUP protocols. That is, they deal with sending and receiving PUP packets. It is assumed the reader is familiar with the format and use of pups, e.g., from reading reference [3] in section 21.1.7.

(RESTART.ETHER) [Function]

This function is intended to be invoked from the executive on those rare occasions when the Ethernet appears completely unresponsive, due to Lisp having gotten into a bad state. RESTART.ETHER reinitializes Lisp's Ethernet driver(s), just as when the Lisp system is started up following a LOGOUT, SYSOUT, etc. This aborts any Ethernet activity and clears several internal caches, including the routing table.

21.5.1 Creating and Managing Pups

There is a record PUP that overlays the data portion of an ETHERPACKET and describes the format of a pup. This record defines the following numeric fields: PUPLENGTH (16 bits), TCONTROL (transmit control, 8 bits, cleared when a PUP is transmitted), PUPTYPE (8 bits), PUPID (32 bits), PUPIDHI and PUPIDLO (16 bits each overlaying PUPID), PUPDEST (16 bits overlayed by 8-bit fields PUPDESTNET and PUPDESTHOST), PUPDESTSOCKET (32 bits, overlayed by 16-bit fields PUPDESTSOCKETHI and PUPDESTSOCKETLO), and PUPSOURCE, PUPSOURCENET, PUPSOURCEHOST, PUPSOURCESOCKET, PUPSOURCESOCKETHI, and PUPSOURCESOCKETLO, analagously. The field PUPCONTENTS is a pointer to the start of the data portion of the pup.

(ALLOCATE.PUP) [Function]

Returns a (possibly used) pup. Keeps a free pool, creating new pups only when necessary. The pup header fields of the pup returned are guaranteed to be zero, but there may be garbage in the data portion if the pup had been recycled, so the caller should clear the data if desired.

(CLEARPUP *PUP*) [Function]

Clears *all* information from *PUP*, including the pointer fields of the ETHERPACKET and the pup data portion.

(RELEASE.PUP *PUP*) [Function]

Releases *PUP* to the free pool.

21.5.2 Sockets

Pups are sent and received on a *socket*. Generally, for each "conversation" between one machine and another, there is a distinct socket. When a pup arrives at a machine, the low-level pup software examines the pup's destination socket number. If there is a socket on the machine with that number, the incoming pup is handed over to the socket; otherwise the incoming pup is discarded. When a *user* process initiates a conversation, it generally selects a large, random socket number different from any other in use on the machine. A *server* process, on the other hand, provides a specific service at a "well-known" socket, usually a fairly small number. In the PUP world, advertised sockets are in the range 0 to 100Q.

(OPENPUPSOCKET *SKT# IFCLASH*) [Function]

Opens a new pup socket. If *SKT#* is NIL (the normal case), a socket number is chosen automatically, guaranteed to be unique, and probably different from any socket opened this way in the last 18 hours (the low half of the time of day clock is sampled).

If a specific local socket is desired, as is typically the case when implementing a server, *SKT#* is given, and must be a (up to 32-bit) number. *IFCLASH* indicates what to do in the case that the designated socket is already in use: if NIL, an error is generated; if ACCEPT, the socket is quietly returned; if FAIL, then OPENPUPSOCKET returns NIL without causing an error. Note that "well-known" socket numbers should be avoided unless the caller is actually implementing one of the services advertised as provided at the socket.

(CLOSEPUPSOCKET *PUPSOC NOERRORFLG*) [Function]

Closes and releases socket *PUPSOC*. If *PUPSOC* is T, closes all pup sockets (this must be used with caution, since it will also close system sockets!). If *PUPSOC* is already closed, an error is generated unless *NOERRORFLG* is true.

(PUPSOCKETNUMBER *PUPSOC*) [Function]

Returns the socket number (a 32-bit integer) of *PUPSOC*.

(PUPSOCKETEVENT *PUPSOC*) [Function]

Returns the EVENT of *PUPSOC* (page 18.30). This event is notified whenever a pup arrives on *PUPSOC*, so pup clients can perform an AWAIT.EVENT on this event if they have nothing else to do at the moment.

21.5.3 Sending and Receiving Pups

(SENDPUP *PUPSOC PUP*) [Function]

Sends *PUP* on socket *PUPSOC*. If any of the PUPSOURCESHOST, PUPSOURCENET, or PUPSOURCESOCKET fields is zero, SENDPUP fills them in using the pup address of this machine and/or the socket number of *PUPSOC*, as needed.

(GETPUP *PUPSOC WAIT*) [Function]

Returns the next pup that has arrived addressed to socket *PUPSOC*. If there are no pups waiting on *PUPSOC*, then GETPUP returns NIL, or waits for a pup to arrive if *WAIT* is T. If *WAIT* is an integer, GETPUP interprets it as a number of milliseconds to wait, finally returning NIL if a pup does not arrive within that time.

(DISCARDPUPS *SOC*) [Function]

Discards without examination any pups that have arrived on *SOC* and not yet been read by a GETPUP.

(EXCHANGEPUPS *SOC OUTPUP DUMMY IDFILTER TIMEOUT*) [Function]

Sends *OUTPUP* on *SOC*, then waits for a responding pup, which it returns. If *IDFILTER* is true, ignores pups whose PUPID is different from that of *OUTPUP*. *TIMEOUT* is the length of time (msecs) to wait for a response before giving up and returning NIL. *TIMEOUT* defaults to \ETHERTIMEOUT. EXCHANGEPUPS discards without examination any pups that are currently waiting on *SOC* before *OUTPUP* gets sent. (*DUMMY* is ignored; it exists for compatibility with an earlier implementation).

21.5.4 Pup Routing Information

Ordinarily, a program calls SENDPUP and does not worry at all about the route taken to get the pup to its destination. There is an internet routing process in Lisp whose job it is to maintain information about the best routes to networks of interest. However, there are some algorithms for which routing information and/or the topology of the net are explicitly desired. To this end, the following functions are supplied:

(PUPNET.DISTANCE *NET#*) [Function]

Returns the "hop count" to network *NET#*, i.e., the number of gateways through which a pup must pass to reach *NET#*, according to the best routing information known at this point. The local (directly-connected) network is considered to be zero hops away. Current convention is that an inaccessible network is 16 hops away. PUPNET.DISTANCE may need to wait to obtain routing information from an Internetwork Router if *NET#* is not currently in its routing cache.

(SORT.PUPHOSTS.BY.DISTANCE *HOSTLIST*) [Function]

Sorts *HOSTLIST* by increasing distance, in the sense of PUPNET.DISTANCE. *HOSTLIST* is a list of lists, the CAR of each list being a 16-bit Net/Host address, such as returned by ETHERHOSTNUMBER. In particular, a list of ports ((nethost . socket) pairs) is in this format.

(PRINTROUTINGTABLE *TABLE SORT FILE*) [Function]

Prints to *FILE* the current routing cache. The table is sorted by network number if *SORT* is true. *TABLE* = PUP (the default) prints the PUP routing table; *TABLE* = NS prints the NS routing table..

21.5.5 Miscellaneous PUP Utilities

(SETUPPUP *PUP DESTHOST DESTSOCKET TYPE ID SOC REQUEUE*) [Function]

Fills in various fields in *PUP*'s header: its length (the header overhead length; assumes data length of zero), *TYPE*, *ID* (if *ID* is NIL, generates a new one itself from an internal 16-bit counter), destination host and socket (*DESTHOST* may be anything that ETHERPORT accepts; an explicit nonzero socket in *DESTHOST* overrides *DESTSOCKET*). If *SOC* is not supplied, a new socket is opened. *REQUEUE* fills the packets EPREQUEUE field (see above). Value of SETUPPUP is the socket.

(SWAPPUPPORTS *PUP*) [Function]

Swaps the source and destination addresses in *PUP*. This is useful in simple packet exchange protocols, where you want to respond to an input packet by diddling the data portion and then sending the pup back whence it came.

(GETPUPWORD *PUP WORD#*) [Function]

Returns as a 16-bit integer the contents of the *WORD#*th word of *PUP*'s data portion, counting the first word as word zero.

(PUTPUPWORD *PUP WORD# VALUE*) [Function]

Stores 16-bit integer *VALUE* in the *WORD#*th word of *PUP*'s data portion.

(GETPUPBYTE *PUP BYTE#*) [Function]

Returns as an integer the contents of the *BYTE#*th 8-bit byte of *PUP*'s data portion, counting the first byte as byte zero.

(PUTPUPBYTE *PUP BYTE# VALUE*) [Function]

Stores *VALUE* in the *BYTE#*th 8-bit byte of *PUP*'s data portion.

(GETPUPSTRING *PUP OFFSET*) [Function]

Returns a string consisting of the characters in *PUP*'s data portion starting at byte *OFFSET* (default zero) through the end of *PUP*.

(PUTPUPSTRING *PUP STR*) [Function]

Appends *STR* to the data portion of *PUP*, incrementing *PUP*'s length appropriately.

21.5.6 PUP Debugging Aids

Tracing facilities are provided to allow the user to see the pup traffic that passes through SENDPUP and GETPUP. The tracing can be verbose, displaying much information about each packet, or terse, which shows a concise "picture" of the traffic.

PUPTRACEFLG [Variable]

Controls tracing information provided by SENDPUP and GETPUP. Legal values:

NIL No tracing.

T Every SENDPUP and every successful GETPUP call PRINTPUP of the pup at hand (see below).

PEEK Allows a concise "picture" of the traffic. For normal, non-broadcast packets, SENDPUP prints "!", GETPUP prints "+". For broadcast packets, SENDPUP prints "↑", GETPUP prints "*". In addition, for packets that arrive not addressed to any socket on this machine (e.g., broadcast packets for a service not implemented on this machine), a "&" is printed.

PUPIGNORETYPES [Variable]

A list of pup types (small integers). If the type of a pup is on this list, then GETPUP and SENDPUP will not print the pup verbosely, but treat it as though PUPTRACEFLG were PEEK. This allows the user to filter out "uninteresting" pups, e.g., routine routing information pups (type 201Q).

PUPONLYTYPES [Variable]

A list of pup types. If this variable is non-NIL, then GETPUP and SENDPUP print verbosely *only* pups whose types appear on the list, treating others as though PUPTRACEFLG were PEEK. This lets the tracing be confined to only a certain class of pup traffic.

PUPTRACEFILE [Variable]

The file to which pup tracing output is sent by default. The file must be open. PUPTRACEFILE is initially T.

PUPTRACETIME [Variable]

If this variable is true, then each printout of a pup is accompanied by a relative timestamp (in seconds, with 2 decimal places) of the current time (i.e., when the SENDPUP or GETPUP was called; for incoming pups, this is not the same as when the pup actually arrived).

(PUPTRACE *FLG REGION*) [Function]

Creates a window for puptracing, and sets PUPTRACEFILE to it. If PUPTRACEFILE is currently a window and *FLG* is NIL, closes the window. Sets PUPTRACEFLG to be *FLG*. If *REGION* is supplied, the window is created with that region. The window's BUTTONEVENTFN is set to cycle PUPTRACEFLG through the values NIL, T, and PEEK when the mouse is clicked in the window.

(PRINTPUP *PACKET CALLER FILE PRE.NOTE DOFILTER*) [Function]

Prints the information in the header and possibly data portions of pup *PACKET* to *FILE*. If *CALLER* is supplied, it identifies the direction of the pup (GET or PUT), and is printed in front of the header. *FILE* defaults to PUPTRACEFILE. If *PRE.NOTE* is non-NIL, it is PRIN1'ed first. If *DOFILTER* is true, then if *PUP*'s type fails the filtering criteria of PUPIGNORETYPES or PUPONLYTYPES, then *PUP* is printed "tersely", i.e., as a !, +, ↑, or *, as described above.

GETPUP and SENDPUP, when PUPTRACEFLG is non-NIL, call (PRINTPUP *PUP* {'GET or 'PUT} NIL NIL T).

The form of printing provided by PRINTPUP can be influenced by adding elements to PUPPRINTMACROS.

PUPPRINTMACROS [Variable]

An association list of elements (*PUPTYPE* . *MACRO*) for printing pups. The *MACRO* (CDR of each element) tells how to print the information in a pup of type *PUPTYPE* (CAR of the element). If *MACRO* is a litatom, then it is a function of two arguments (*PUP FILE*) that is applied to the pup to do the printing. Otherwise, *MACRO* is a list describing how to print the data portion of the pup (the header is printed in a standard way).

The list form of *MACRO* consists of "commands" that specify a "datatype" to interpret the data, and an indication of how far that datatype extends in the packet. Each element of *MACRO* is one of the following: (a) a byte offset (positive integer), indicating the byte at which the next element, if any, takes effect; (b) a negative integer, the absolute value of which is the number of bytes until the next element, if any, takes effect; or (c) an atom giving the format in which to print the data, one of the following:

BYTES — Print the data as 8-bit bytes, enclosed in brackets. This is the default format to start with.

CHARS — Print the data as (8-bit) characters. Non-printing characters are printed as if the format were BYTES, except that the sequence 15Q, 12Q is printed specially as [crlf].

WORDS — Print the data as 16-bit integers, separated by commas (or the current SEPR).

INTEGERS

Print the data as 32-bit integers, separated by commas (or the current SEPR). Note: the singular BYTE, CHAR, WORD, INTEGER are accepted as synonyms for these four commands.

SEPR

Set the separator for WORDS and INTEGERS to be the next element of the macro. The separator is initially the two characters, comma, space.

IFSSTRING

Interprets the data as a 16-bit length followed by that many 8-bit bytes or characters. If the current datatype is BYTES, leaves it alone; otherwise, sets it to be CHARS.

...

If there is still data left in the packet by the time processing reaches this command, prints "···" and stops.

FINALLY

The next element of the macro is printed when the end of the packet is reached (or printing stops because of a ···). This command does not alter the datatype, and can appear anywhere in the macro as long as it is encountered before the actual end of the packet.

T

Perform a TERPRI.

REPEAT

The remainder of the macro is itself treated as a macro to be applied over and over until the packet is exhausted. Note that the offsets specified in the macro must be in the relative form, i.e., negative integers. For example, the macro (INTEGERS 4 REPEAT BYTES -2 WORDS -4) says to print the first 4 bytes of the data as one 32-bit integer, then print the rest of the data as sets of 2 8-bit bytes and 2 16-bit words.

Only as much of the macro is processed as is needed to print the data in the given packet. The default macro for printing a pup is (BYTES 12 ...), meaning to print the first up to 12 bytes as bytes, and then print "···" if there is anything left.

The following functions are used by PRINTPUP and similar functions, and may be of interest in special cases.

(PORTSTRING *NETHOST SOCKET*) [Function]

Converts the pup address *NETHOST*, *SOCKET* into the following octal string format: net#host#socket. *NETHOST* may be a port (dotted pair of nethost and socket), in which case *SOCKET* is ignored, and the socket portion of *NETHOST* is omitted from the string if it is zero.

(PRINTPUPROUTE *PACKET CALLER FILE*) [Function]

Prints the source and destination addresses of pup *PACKET* to *FILE* in the PORTSTRING format, preceded by *CALLER* (interpreted as with PRINTPUP).

(PRINTPACKETDATA *BASE OFFSET MACRO LENGTH FILE*) [Function]

Prints data according to *MACRO*, which is a list interpreted as described under

PUPPRINTMACROS, to *FILE*. The data starts at *BASE* and extends for *LENGTH* bytes. The actual printing starts at the *OFFSET*th byte, which defaults to zero. For example, PRINTPUP ordinarily calls (PRINTPACKETDATA (fetch PUPCONTENTS of *PUP*) 0 *MACRO* (IDIFFERENCE (fetch PUPLENGTH of *PUP*) 20) *FILE*).

(PRINTCONSTANT *VAR CONSTANTLIST FILE PREFIX*) [Function]

CONSTANTLIST is a list of pairs (*VARNAME VALUE*), of the form given to the CONSTANTS File Package Command. PRINTCONSTANT prints *VAR* to *FILE*, followed in parentheses by the *VARNAME* out of *CONSTANTLIST* whose *VALUE* is EQ to *VAR*, or ? if it finds no such element. If *PREFIX* is non-NIL and is an initial substring of the selected *VARNAME*, then *VARNAME* is printed without the prefix.

For example, if FOOCONSTANTS is ((FOO.REQUEST 1) (FOO.ANSWER 2) (FOO.ERROR 3)), then (PRINTCONSTANT 2 FOOCONSTANTS T "FOO.") produces "2 (ANSWER)".

(OCTALSTRING *N*) [Function]

Returns a string of octal digits representing *N* in radix 8.

21.6 NS LEVEL ONE FUNCTIONS

The functions in this section are used to implement level two and higher NS protocols. The packets used in the NS protocol are termed Xerox Internet Packets (XIPs). The functions for manipulating XIPs are similar to those for managing PUPs, so will be described in less detail here. The major difference is that NS host addresses are 48-bit numbers. Since Interlisp-D cannot currently represent 48-bit numbers directly as integers, there is an interim form called NSHOSTNUMBER, which is defined as a TYPERECORD of three fields, each of them being a 16-bit portion of the 48-bit number.

21.6.1 Creating and Managing XIPs

There is a record XIP that overlays the data portion of an ETHERPACKET and describes the format of a XIP. This record defines the following fields: XIPLENGTH (16 bits), XIPTCONTROL (transmit control, 8 bits, cleared when a XIP is transmitted), XIPTYPE (8 bits), XIPDESTNET (32 bits), XIPDESTHOST (an NSHOSTNUMBER), XIPDESTSOCKET (16 bits), and XIPSOURCENET, XIPSOURCEHOST, and XIPSOURCESOCKET, analagously. The field XIPCONTENTS is a pointer to the start of the data portion of the XIP.

(ALLOCATE.XIP) [Function]

Returns a (possibly used) XIP. As with ALLOCATE.PUP, the header fields are guaranteed to be zero, but there may be garbage in the data portion if the pup had been recycled.

(RELEASE.XIP *XIP*) [Function]

Releases *XIP* to the free pool.

21.6.2 NS Sockets

As with pups, XIPs are sent and received on a *socket.* The same comments apply as with pup sockets (page 21.16), except that NS socket numbers are only 16 bits.

(OPENNSOCKET *SKT# IFCLASH*) [Function]

Opens a new NS socket. If *SKT#* is NIL (the normal case), a socket number is chosen automatically, guaranteed to be unique, and probably different from any socket opened this way in the last 18 hours. If a specific local socket is desired, as is typically the case when implementing a server, *SKT#* is given, and must be a (up to 16-bit) number. *IFCLASH* governs what to do if *SKT#* is already in use, as with OPENPUPSOCKET.

(CLOSENSOCKET *NSOC NOERRORFLG*) [Function]

Closes and releases socket *NSOC*. If *NSOC* is T, closes all NS sockets (this must be used with caution, since it will also close system sockets!). If *NSOC* is already closed, an error is generated unless *NOERRORFLG* is true.

(NSOCKETNUMBER *NSOC*) [Function]

Returns the socket number (a 16-bit integer) of *NSOC*.

(NSOCKETEVENT *NSOC*) [Function]

Returns the EVENT of *NSOC*. This event is notified whenever a XIP arrives on *NSOC*.

21.6.3 Sending and Receiving XIPs

(SENDXIP *NSOC XIP*) [Function]

Sends *XIP* on socket *NSOC*. If any of the XIPSOURCESHOST, XIPSOURCENET, or XIPSOURCESOCKET fields is zero, SENDXIP fills them in using the NS address of this machine and/or the socket number of *NSOC*, as needed.

(GETXIP *NSOC WAIT*) [Function]

Returns the next XIP that has arrived addressed to socket *NSOC*. If there are no XIPs waiting on *NSOC*, then GETXIP returns NIL, or waits for a XIP to arrive if *WAIT* is T. If *WAIT* is an integer, GETXIP interprets it as a number of milliseconds to wait, finally returning NIL if a XIP does not arrive within that time.

(DISCARDXIPS *NSOC*) [Function]

Discards without examination any XIPs that have arrived on *NSOC* and not yet been read by a GETXIP.

(EXCHANGEXIPS *SOC OUTXIP IDFILTER TIMEOUT*) [Function]

Useful for simple NS packet exchange protocls. Sends *OUTXIP* on *SOC*, then waits for a responding XIP, which it returns. If *IDFILTER* is true, ignores XIPs whose packet exchange ID (the first 32 bits of the data portion) is different from that of *OUTXIP*. *TIMEOUT* is the length of time (msecs) to wait for a response before giving up and returning NIL. *TIMEOUT* defaults to \ETHERTIMEOUT. EXCHANGEXIPS discards without examination any XIPs that are currently waiting on *SOC* before *OUTXIP* gets sent.

21.6.4 NS Debugging Aids

XIPs can be printed automatically by SENDXIP and GETXIP analogously to the way pups are. The following variables behave with respect to XIPs the same way that the corresponding PUP-named variables behave with respect to PUPs: XIPTRACEFLG, XIPTRACEFILE, XIPIGNORETYPES, XIPONLYTYPES, XIPPRINTMACROS. In addition, the functions PRINTXIP, PRINTXIPROUTE and XIPTRACE are directly analogous to PRINTPUP, PRINTPUPROUTE, and PUPTRACE.

21.7 SUPPORT FOR OTHER LEVEL ONE PROTOCOLS

Raw packets other than of type PUP or NS can also be sent and received. This section describes facilities to support such protocols. Many of these functions have a \ in their names to designate that they are system internal, not to be dealt with as casually as user-level functions.

(\ALLOCATE.ETHERPACKET) [Function]

Returns an ETHERPACKET datum. Enough of the packet is cleared so that if the packet represents a PUP or NS packet, that its header is all zeros; no guarantee is made about the remainder of the packet.

(\RELEASE.ETHERPACKET *EPKT*) [Function]

Returns *EPKT* to the pool of free packets. This operation is dangerous if the caller actually is still holding on to *EPKT*, e.g., in some queue, since this packet could be returned to someone else (via \ALLOCATE.ETHERPACKET) and suffer the resulting contention.

From a logical standpoint, programs need never call \RELEASE.ETHERPACKET, since the packets are eventually garbage-collected after all pointers to them drop. However, since the packets are so large, normal garbage collections tend not to occur frequently enough. Thus, for best performance, a well-disciplined program should explicitly release packets when it knows it is finished with them.

A locally-connected network for the transmission and receipt of Ether packets is specified by a *network descriptor block*, an object of type NDB. There is one NDB for each directly-connected network; ordinarily there is only one. The NDB contains information specific to the network, e.g., its PUP and NS network numbers, and information about how to send and receive packets on it.

\LOCALNDBS [Variable]

The first NDB connected to this machine, or NIL if there is no network. Any other NDBs are linked to this first one via the NDBNEXT field of the NDB.

In order to transmit an Ether packet, a program must specify the packet's type and its immediate destination. The type is a 16-bit integer identifying the packet's protocol. There are preassigned types for PUP and NS. The destination is a host address on the local network, in whatever form the local network uses for addressing; it is not necessarily related to the logical ultimate destination of the packet. Determining the immediate destination of a packet is the task of *routing*. The functions SENDPUP and SENDXIP take care of this for the PUP and NS protocols, routing a packet directly to its destination if that host is on the local network, or routing it to a gateway if the host is on some other network accessible via the gateway. Of course, a gateway must know about the type (protocol) of a packet in order to be able to forward it.

(ENCAPSULATE.ETHERPACKET *NDB PACKET PDH NBYTES ETYPE*) [Function]

Encapsulates *PACKET* for transmission on network *NDB*. *PDH* is the physical destination host (e.g., an 8-bit pup host number or a 48-bit NS host number); *NBYTES* is the length of the packet in bytes; *ETYPE* is the packet's encapsulation type (an integer).

(TRANSMIT.ETHERPACKET *NDB PACKET*) [Function]

Transmits *PACKET*, which must already have been encapsulated, on network *NDB*. Disposition of the packet after transmission is complete is determined by the value of *PACKET*'s EPREQUEUE field.

In order to receive Ether packets of type other than PUP or NS, the programmer must specify what to do with incoming packets. Lisp maintains a set of *packet filters*, functions whose job it is to appropriately dispose of incoming packets of the kind they want. When a packet arrives, the Ethernet driver calls each filter function in turn until it finds one that accepts the packet. The filter function is called with two arguments: (*PACKET TYPE*), where *PACKET* is the actual packet, and *TYPE* is its Ethernet encapsulation type (a number). If a filter function accepts the packet, it should do what it wants to with it, and return T; else it should return NIL, allowing other packet filters to see the packet.

Since the filter function is run at interrupt level, it should keep its computation to a minimum. For example, if there is a lot to be done with the packet, the filter function can place it on a queue and notify another process of its arrival.

The system already supplies packet filters for packets of type PUP and NS; these filters enqueue the incoming packet on the input queue of the socket to which the packet is addressed, after checking that the packet is well-formed and indeed addressed to an existing socket on this machine.

Incoming packets have their EPNETWORK field filled in with the NDB of the network on which the packet arrived.

(\ADD.PACKET.FILTER *FILTER*) [Function]

Adds function *FILTER* to the list of packet filters if it is not already there.

(\DEL.PACKET.FILTER *FILTER*) [Function]

Removes *FILTER* from the list of packet filters.

(\CHECKSUM *BASE NWORDS INITSUM*) [Function]

Computes the one's complement add and cycle checksum for the *NWORDS* words starting at address *BASE*. If *INITSUM* is supplied, it is treated as the accumulated checksum for some set of words preceding *BASE*; normally *INITSUM* is omitted (and thus treated as zero).

(PRINTPACKET *PACKET CALLER FILE PRE.NOTE DOFILTER*) [Function]

Prints *PACKET* by invoking a function appropriate to *PACKET*'s type. See PRINTPUP for the intended meaning of the other arguments. In order for PRINTPACKET to work on a non-standard packet, there must be information on the list \PACKET.PRINTERS.

\PACKET.PRINTERS [Variable]

An association list mapping packet type into the name of a function for printing that type of packet.

21.8 THE SYSQUEUE MECHANISM

The SYSQUEUE facility provides a low-level queueing facility. The functions described herein are all system internal: they can cause much confusion if misused.

A SYSQUEUE is a datum containing a pointer to the first element of the queue and a pointer to the last; each item in the queue points to the next via a pointer field located at offset 0 in the item (its QLINK field in the QABLEITEM record). A SYSQUEUE can be created by calling (NCREATE 'SYSQUEUE).

(\ENQUEUE *Q ITEM*) [Function]

Enqueues *ITEM* on *Q*, i.e., links it to the tail of the queue, updating *Q*'s tail pointer appropriately.

(\DEQUEUE *Q*) [Function]

Removes the first item from *Q* and returns it, or returns NIL if *Q* is empty.

(\UNQUEUE *Q ITEM NOERRORFLG*) [Function]

Removes the *ITEM* from *Q*, wherever it is located in the queue, and returns it. If *ITEM* is not in *Q*, causes an error, unless *NOERRORFLG* is true, in which case it returns NIL.

(\QUEUELENGTH *Q*) [Function]

Returns the number of elements in *Q*.

(\ONQUEUE *ITEM Q*) [Function]

True if *ITEM* is an element of *Q*.

The SYSQUEUE mechanism

CHAPTER 22

INTERLISP-10 SPECIFICS

This chapter describes a number of features of Interlisp-10 that are machine or implementation-dependent, and are not expected to be implemented in newer implementations of Interlisp.

22.1 INTERLISP-10 INTERRUPT CHARACTERS

The table below gives the interrupt characters currently enabled in Interlisp-10.

Note: It is possible to change the assignments of control characters to interrupts with INTERRUPTCHAR (page 9.17).

control-B Generates an immediate error, and causes a break, regardless of the depth or time of the computation. Thus if the function FOO is looping internally, typing control-B will cause the computation to be stopped, the stack unwound to the point at which FOO was called, and then cause a break.

This is a stronger interruption than control-H. Note that the internal variables of FOO above are not available in this break, and similarly, FOO may have already produced some changes in the environment before the control-B was typed. It may not be possible to simply continue the computation, depending on the nature of the function interrupted and when it was interrupted. Therefore whenever possible, it is better to use control-H instead of control-B.

control-C Computation is stopped, and control returns to the operating system (Tenex, etc.) The program can be continued with the CONTINUE command.

control-D Aborts the computation, and unwinds the stack to the top level. Calls RESET (page 9.14).

control-E Aborts the computation, and unwinds the stack to the last ERRORSET. Calls ERROR! (page 9.14).

control-H At the next point a function is about to be entered, the function INTERRUPT is called instead. INTERRUPT types INTERRUPTED BEFORE *FN*, constructs an appropriate break expression, and then calls BREAK1. The user can then examine the state of the computation, and continue by typing OK, GO or EVAL, and/or RETFROM back to some previous point, exactly as with a user break. Control-H breaks are thus always "safe".

Control-H breaks *only* occur when a function is called, since it is only at this time that the system is in a "clean" enough state to allow the user to interact. Thus, if a compiled program is looping without calling any functions (or if Interlisp-10 is in a I/O wait), control-H will not affect it. Control-B, however, will.

As soon as control-H is typed, Interlisp clears and saves the input buffer, and then rings the bell, indicating that it is now safe to type ahead to the upcoming break. If the break returns a value, i.e., is not aborted via ↑ or control-D, the contents of the input buffer before the control-H was typed will be restored.

Note: Control-H will *not* interrupt at linked function calls (see page 12.18).

control-O Clears the teletype output buffer.

control-P Changes the `PRINTLEVEL` setting (see page 6.18).

control-S Changes the `MINFS` setting (see page 22.10).

control-T Prints total execution time for the program, as well as other status information.

22.2 TYPE NUMBER FUNCTIONS

Each data type in Interlisp has an associated "type name". In Interlisp-10, each data type also has a "type number", which can be accessed and manipulated with the functions below. In general, it is preferable to use the type name functions (see page 2.1).

`(NTYP` *DATUM*`)` [Function]

Returns the type number for the data type of *DATUM*, e.g., `(NTYP '(A . B))` is 8, the type number for lists.

`(TYPEP` *DATUM N*`)` [Function]

Value is `T`, if the type number of *DATUM* is equal to *N*.

`(TYPENAMEFROMNUMBER` *N*`)` [Function]

Value is type name for type number *N*, or `NIL` if *N* is not a valid type number, e.g. `(TYPENAMEFROMNUMBER 30)=STRING.CHARS`.

`(TYPENUMBERFROMNAME` *NAME*`)` [Function]

Value is corresponding type number for *NAME*, or `NIL` if *NAME* is not a type name, e.g. `(TYPENUMBERFROMNAME 'STRING.CHARS)=30`.

`TYPENUMBERFROMNAME` will accept `READTABLEP`, `TERMTABLEP`, `CCODEP`, and `ARRAYP`, and return the same value for each, which for Interlisp-10 is 1. Note however that `(TYPENAMEFROMNUMBER 1)=ARRAYP`.

`(GETTYPEDESCRIPTION` *TYPE*`)` [Function]

Returns the type description string for *TYPE*, a type name or type number.

`(SETTYPEDESCRIPTION` *TYPE STRING*`)` [Function]

Sets the type description string for *TYPE* to be *STRING*. The type description is used in garbage collection messages and by `STORAGE`.

22.3 VALIDITY OF DEFINITIONS IN INTERLISP-10

Although the function definition cell is intended for function definitions, PUTD and GETD do not make thorough checks on the validity of definitions that "look like" exprs, compiled code, or SUBRs. Thus if PUTD is given an array pointer, it treats it as compiled code, and simply stores the array pointer in the definition cell. GETD will then return the array pointer. Similarly, a call to that function will simply transfer to what would normally be the entry point for the function, and produce random results if the array were not compiled function.

Similarly, if PUTD is given a dotted pair of the form (number . address) where number and address fall in the subr range, PUTD assumes it is a subr and stores it away as described earlier. GETD would then return a dotted pair EQUAL (but not EQ) to the expression originally given PUTD. Similarly, a call to this function would transfer to the corresponding address.

Finally, if PUTD is given any other list, it simply stores it away. A call to this function would then go through the interpreter.

Note that PUTD does not actually check to see if the s-expression is valid definition, i.e., begins with LAMBDA or NLAMBDA. Similarly, EXPRP is true if a definition is a list and not of the form (number . address), number = 0, 1, 2, or 3 and address a subr address; SUBRP is true if it is of this form. ARGLIST and NARGS work correspondingly.

Only FNTYP and ARGTYPE check function definitions further than that described above: both ARGTYPE and FNTYP return NIL when EXPRP is true but CAR of the definition is not LAMBDA or NLAMBDA.[1] In other words, if the user uses PUTD to put (A B C) in a function definition cell, GETD will return this value, the editor and prettyprint will both treat it as a definition, EXPRP will return T, CCODEP and SUBRP NIL, ARGLIST B, and NARGS 1.

22.4 REUSING BOXED NUMBERS IN INTERLISP-10 - SETN

RPLACA and RPLACD provide a way of cannibalizing list structure for reuse in order to avoid making new structure and causing garbage collections.[2] This section describes an analogous function in Interlisp-10 for reusing large integers and floating point numbers, SETN. SETN is used like SETQ, i.e., its first argument is considered as quoted, its second is evaluated. If the current value of the variable being set is a large integer or floating point number, the new value is deposited into that word in number storage, i.e., no new storage is used.[3] If the current value is *not* a large integer or floating point number, e.g., it can be

[1] These functions have different values on LAMBDAs and NLAMBDAs and hence must check. The compiler and interpreter also take different actions for LAMBDAs and NLAMBDAs, and therefore generate errors if the definition is neither.

[2] The nobox package provides a more aesthetic way of reusing cons cells as well as number boxes. However, it is still the case that techniques involving reusing static storage should be used with extreme caution, and be reserved for those cases where the normal method of storage allocation and garbage collection is not workable or practical. The decl package (page 23.18) takes a different approach to the same problem by avoiding creating number boxes in the first place via type declarations in the body of the function definition.

[3] The second argument to SETN must always be a number or a NON-NUMERIC ARG error is generated.

NIL, SETN operates exactly like SETQ, i.e., the large integer or floating point number is boxed, and the variable is set. This eliminates initialization of the variable.

SETN will work interpretively, i.e., reuse a word in number storage, but will not yield any savings of storage because the boxing of the second argument will still take place, when it is evaluated. The elimination of a box is achieved only when the call to SETN is compiled, since SETN compiles open, and does not perform the box if the old value of the variable can be reused.

22.4.1 Caveats concerning use of SETN

There are three situations to watch out for when using SETN. The first occurs when the same variable is being used for floating point numbers and large integers. If the current value of the variable is a floating point number, and it is reset to a large integer, via SETN, the large integer is simply deposited into a word in floating point number storage, and hence will be interpreted as a floating point number. Thus,

```
←(SETQ FOO 2.3)
2.3
←(SETN FOO 10000)
2.189529E-43
```

Similarly, if the current value is a large integer, and the new value is a floating point number, equally strange results occur.

The second situation occurs when a SETN variable is reset from a large integer to a small integer. In this case, the small integer is simply deposited into large integer storage. It will then print correctly, and function arithmetically correctly, but it is *not* a small integer, and hence will not be EQ to another integer of the same value, e.g.,

```
←(SETQ FOO 10000)
10000
←(SETN FOO 1)
1
←(IPLUS FOO 5)
6
←(EQ FOO 1)
NIL
←(SMALLP FOO)
NIL
```

In particular, note that ZEROP will return NIL even if the variable is equal to 0. Thus a program which begins with FOO set to a large integer and counts it down by (SETN FOO (SUB1 FOO)) must terminate with (EQP FOO 0), not (ZEROP FOO).

Finally, the third situation to watch out for occurs when you want to save the current value of a SETN variable for later use. For example, if FOO is being used by SETN, and the user wants to save its current value on FIE, (SETQ FOO FIE) is not sufficent, since the next SETN on FOO will also change FIE, because its changes the word in number storage pointed to by FOO, and hence pointed to by FIE. The number must be copied, e.g., (SETQ FIE (IPLUS FOO)), which sets FIE to a new word in number storage.

(SETN *VAR* *X*) [NLambda Function]

A nlambda function like SETQ. *VAR* is quoted, *X* is evaluated, and its value must be a number. *VAR* will be set to this number. If the current value of *VAR* is a large integer or floating point number, that word in number storage is cannibalized. The value of SETN is the (new) value of *VAR*.

22.5 BOX AND UNBOX IN INTERLISP-10

Some applications may require that a user program explicitly perform the boxing and unboxing operations that are usually implicit (and invisible) to most programs. The functions that perform these operations are LOC and VAG respectively. For example, if a user program executes a TENEX JSYS using the ASSEMBLE directive, the value of the ASSEMBLE expression will have to be boxed to be used arithmetically, e.g., (IPLUS X (LOC (ASSEMBLE --))). It must be emphasized that

Arbitrary unboxed numbers should NOT be passed around as ordinary values because they can cause trouble for the garbage collector.

For example, suppose the value of X were 150000, and you created (VAG X), and this just *happened* to be an address on the free storage list. The next garbage collection could be disastrous. For this reason, the function VAG must be used with extreme caution when its argument's range is not known.

LOC is the inverse of VAG. It takes an address, i.e., a 36 bit quantity, and treats it as a number and boxes it. For example, LOC of an atom, e.g., (LOC (QUOTE FOO)), treats the atom as a 36 bit quantity, and makes a number out of it. If the address of the atom FOO were 125000, (LOC (QUOTE FOO)) would be 125000, i.e., the location of FOO. It is for this reason that the box operation is called LOC, which is short for location.

Note that FOO does not print as #364110 (125000 in octal) because the print routine recognizes that it is an atom, and therefore prints it in a special way, i.e., by printing the individual characters that comprise it. Thus (VAG 125000) would print as FOO, and would in fact *be* FOO.

(LOC *X*) [Function]

Makes a number out of *X*, i.e., returns the location of *X*.

(VAG *X*) [Function]

The inverse of LOC. *X* must be a number; the value of VAG is the unbox of *X*.

The compiler eliminates extra VAG's and LOC's for example (IPLUS X (LOC (ASSEMBLE --))) will not box the value of the ASSEMBLE, and then unbox it for the addition.

22.6 MISCELLANEOUS OPERATING SYSTEM FUNCTIONS

(LOADAV) [Function]

Returns the current load average as a floating point number (this number is the first of the three printed by the SYSTAT command).

(ERSTR *ERN* —) [Function]

ERN is an error number from a JSYS fail return. *ERN*=NIL means the most recent error. ERSTR returns the operating system error diagnostic as a string.

(JSYS *N AC1 AC2 AC3 RESULTAC*) [Function]

Loads the (unboxed) values of *AC1*, *AC2*, and *AC3* into appropriate accumulaters, and executes JSYS number *N*. If *AC1*, *AC2*, or *AC3*=NIL, 0 is used. JSYS returns the (boxed) contents of the accumulator specified by *RESULTAC*, i.e., 1 means *AC1*, 2 means *AC2*, and 3 means *AC3*, with NIL equivalent to 1. Compiles open if *N* is itself a small integer, and *RESULTAC* is a small integer, or NIL.

If the JSYS causes a trap, the message TRAP AT LOCATION *NNNNN* is printed by the operating system, followed by JSYS ERROR: and the operating system diagnostic. The user is then talking to the operating system exactly as though control-C had been typed. If the user then continues using the CONTINUE command, an Interlisp error is generated, JSYS ERROR, and control then proceeds the same as for any other flavor of error, i.e. unwinds to last ERRORSET or goes into a break as described on page 9.10.

The CJSYS package (page 23.53) enables calling JSYSes by their corresponding name, rather than their number.

(USERNUMBER *A FLG*) [Function]

If *A*=NIL, returns the login user number; if *A*=T, returns the connected user number; if *A* is a literal atom or string, USERNUMBER returns the number of the corresponding user, or NIL if no such user exists.

On TOPS-20, there is a difference between the user number, which is associated with the job, and the directory number, which is associated with the file system. Therefore, on TOPS-20, if *FLG*=T, USERNUMBER returns the directory number rather than the user number.

(HOSTNAME *HOSTN FLG*) [Function]

Returns the hostname as a string for host number *HOSTN*, e.g. "PARC-MAXC2", "BBN-TENEXD", etc. If *HOSTN*=NIL, the local host is used. If the local host is not an arpanet host, value is NIL. Also returns NIL if *HOSTN* is not a valid host number.

FLG is interpreted the same as in USERNAME.

(HOSTNUMBER) [Function]

Returns the host numer of the local host, or NIL, if the local host is not an arpanet host.

(TENEX *STR FILEFLG*) [Function]

Starts up a lower exec (without a message) using SUBSYS, and then if *FILEFLG*=NIL unreads *STR*, followed by "QUIT"[4] (using BKSYSBUF, page 6.47). TENEX returns

[4] "POP" for Interlisp on TOPS-20.

T if all of *STR* is actually processed/read by the lower exec, NIL if the user control-C's and manually QUIT's back to Interlisp.

If *FILEFLG*=T, TENEX passes the string as the second argument to SUBSYS, instead of unreading it. This has the advantage that *STR* can be of any length, and also that typeahead will not interfere with the call to the lower exec. The disadvantage is that TENEX cannot tell whether the commands to the lower exec terminated successfully, or were aborted. Thus, if *FILEFLG*=T, the value of TENEX is always T.

For example, LISTFILES (page 11.9) is implemented using TENEX, with *FILEFLG*=NIL, so LISTFILES can tell if listings actually were completed.

22.7 STORAGE ALLOCATION AND GARBAGE COLLECTION

In the following discussion, we will speak of a quantity of memory being assigned to a particular data-type, meaning that the space is reserved for storage of elements of that type. *Allocation* will refer to the process used to obtain from the already assigned storage a particular location for storing one data element.

A small amount of storage is assigned to each data-type when Interlisp-10 is started; additional storage is assigned only during a garbage collection.

The page is the smallest unit of memory that may be assigned for use by a particular data-type. For each page of memory there is a one word entry in a type table. The entry contains the data-type residing on the page as well as other information about the page. The type of a pointer is determined by examining the appropriate entry in the type table.

Storage is allocated as is needed by the functions which create new data elements, such as CONS, PACK, MKSTRING. For example, when a large integer is created by IPLUS, the integer is stored in the next available location in the space assigned to integers. If there is no available location, a garbage collection is initiated, which may result in more storage being assigned.

The storage allocation and garbage collection methods differ for the various data-types. The major distinction is between the types with elements of fixed length and the types with elements of arbitrary length. List cells, atoms, large integers, floating point numbers, and string pointers are fixed length; all occupy 1 word except atoms which use 3 words. Arrays, print names, and strings (string characters) are variable length.

Elements of fixed length types are stored so that they do not overlap page boundaries. Thus the pages assigned to a fixed length type need not be adjacent. If more space is needed, any empty page will be used. The method of *allocating* storage for these types employs a free-list of available locations; that is, each available location contains a pointer to the next available location. A new element is stored at the first location on the free-list, and the free-list pointer is updated.[5]

[5] The allocation routine for list cells is more complicated. Each page containing list cells has a separate free list. First a page is chosen, then the free list for that page is used. Lists are the only data-type which operate this way.

Elements of variable length data-types *are* allowed to overlap page boundaries. Consequently all pages assigned to a particular variable length type must be contiguous. Space for a new element is allocated following the last space used in the assigned block of contiguous storage.

When Interlisp-10 is first called, a few pages of memory are assigned to each data-type. When the allocation routine for a type determines that no more space is available in the assigned storage for that type, a garbage collection is initiated. The garbage collector determines what data is currently in use and reclaims that which is no longer in use. A garbage collection may also be initiated by the user with the function RECLAIM.

Data in use (also called active data) is any data that can be "reached" from the currently running program (i.e., variable bindings and functions in execution) or from atoms. To find the active data the garbage collector "chases" all pointers, beginning with the contents of the push-down lists and the components (i.e., CAR, CDR, and function definition cell) of all atoms with at least one non-trivial component.

When a previously unmarked datum is encountered, it is marked, and all pointers contained in it are chased. Most data-types are marked using bit tables; that is tables containing one bit for each datum. Arrays, however, are marked using a half-word in the array header.

When the mark and chase process is completed, unmarked (and therefore unused) space is reclaimed. Elements of fixed length types that are no longer active are reclaimed by adding their locations to the free-list for that type. This free list allocation method permits reclaiming space without moving any data, thereby avoiding the time consuming process of updating all pointers to moved data. To reclaim unused space in a block of storage assigned to a variable length type, the active elements are compacted toward the beginning of the storage block, and then a scan of all active data that can contain pointers to the moved data is performed to update the pointers.[6]

Whenever a garbage collection of any type is initiated,[7] unused space for all fixed length types is reclaimed since the additional cost is slight. However, space for a variable length type is reclaimed only when that type initiated the garbage collection.

If the amount of storage reclaimed for the type that initiated the garbage collection is less than the minimum free storage requirement for that type, the garbage collector will assign enough additional storage to satisfy the minimum free storage requirement. The minimum free storage requirement for each data may be set with the function MINFS. The garbage collector assigns additional storage to fixed length types by finding empty pages, and adding the appropriate size elements from each page to the free list. Assigning additional storage to a variable length type involves finding empty pages and moving data so that the empty pages are at the end of the block of storage assigned to that type.

In addition to increasing the storage assigned to the type initiating a garbage collection, the garbage collector will attempt to minimize garbage collections by assigning more storage to other fixed length types according to the following algorithm. If the amount of active data of a type has increased since the last garbage collection by more than 1/4 of the MINFS value for that type, storage is increased (if necessary), to attain the MINFS value. If active data has increased by less than 1/4 of the MINFS value,

[6] If Interlisp-10 types the message ARRAYS FOULED during a garbage collection, it means that an array header has been clobbered and no longer makes sense. This can be due to hardware malfunction, or an as yet undiscovered bug in Interlisp. The best thing to do under these circumstances is to give up and start over with a fresh system or sysout.

[7] The "type of a garbage collection" or the "type that initiated a garbage collection" means either the type that ran out of space and called the garbage collector, or the argument to RECLAIM.

available storage is increased to 1/2 MINFS. If there has been no increase, no more storage is added. For example, if the MINFS setting is 2000 words, the number of active words has increased by 700, and after all unused words have been collected there are 1000 words available, 1024 additional words (two pages) will be assigned to bring the total to 2024 words available. If the number of active words had increased by only 300, and there were 500 words available, 512 additional words would be assigned.

(RECLAIM *TYPE*) [Function]

Initiates a garbage collection of type *TYPE*, where *TYPE* is either a type name or type number. Value of RECLAIM is number of words available (for that type) after the collection.

Garbage collections, whether invoked directly by the user or indirectly by need for storage, do not confine their activity solely to the data type for which they were called, but automatically collect some or all of the other types.

(GCGAG *MESSAGE*) [Function]

Affects messages printed by the garbage collector. If *MESSAGE*=T, whenever a garbage collection is begun, "collecting" is printed, followed by the type description of the type that initiated the collection.[8] When the garbage collection is complete, two numbers are printed: the number of words collected for that type, and the total number of words available for that type, i.e., allocated but not necessarily currently in use. Note that other types may also have been collected, and had more storage assigned.

Example:

```
←RECLAIM(18)

collecting large numbers
511, 3071 free cells
3071
←RECLAIM(LITATOM)

collecting atoms
1020, 1020 free cells
1020
```

If *MESSAGE*=NIL, no garbage collection message is printed, either on entering or leaving the garbage collector.

If *MESSAGE* is a list, CAR of *MESSAGE* is printed (using PRIN1) when the garbage collection is begun, and CDR is printed (using PRIN1) when the collection is finished. If *MESSAGE* is a literal atom or string, *MESSAGE* is printed when the garbage collection is begun, and nothing is printed when the collection finishes.

If *MESSAGE* is a number, the message is the same as for (GCGAG T), except if the total number of free pages left after the collection is less than *MESSAGE*, the number of free pages is printed, e.g.,

[8] Note that this type description can be set via the function SETTYPEDESCRIPTION (page 22.2).

```
←GCGAG(100)
T
←RECLAIM()

collecting lists
10369, 10369 free cells, 87 pages left.
```

The initial setting for GCGAG is 40.

The value of GCGAG is its previous setting.

(GCMESS *MESSAGE# STRING*) [Function]

GCGAG is implemented in terms of the primitive GCMESS which can be used to further refine garbage collection messages for specialized applications. The garbage collection message is actually composed of seven separate messages:

collecting large numbers^{12}
511,^{3} 3071 free cells^{4}, 87^{5} pages^{6} left^{7}

message #1 is the "collecting" string. If NIL, then neither it, nor the type dependent field (which is settable via SETTYPEDESCRIPTION described below) is printed.

message #2 is the carriage-return after the type-dependent field. Thus to simply print a string at the beginning of a garbage collection, perform (GCMESS 1) and (GCMESS 2 *STRING*).

message #3 is the "," which comes after the number of cells actually collected. If NIL, then neither it nor that number are printed.

message #4 is the "free cells" which comes after the number of cells that are now allocated. If NIL, neither it nor that number are printed.

message #5 is the number of pages left below which the system prints message 6.

message #6 is the "pages left" message. If NIL, neither it nor the number of pages left are printed.

message #7 is the terminating carriage return.

(MINFS *N TYPE*) [Function]

Sets the minimum amount of free storage which will be maintained by the garbage collector for data types of type number or type name *TYPE*. If, after any garbage collection for that type, fewer than *N* free words are present, sufficient storage will be added (in 512 word chunks) to raise the level to *N*.

If *TYPE*=NIL, LISTP is used, i.e., the MINFS refers to list words.

If *N*=NIL, MINFS returns the current MINFS setting for the corresponding type.

A MINFS setting can also be changed dynamically, even during a garbage collection, by typing control-S[9] followed by a number, followed by a period. When the control-S is typed, Interlisp immediately clears and saves the input buffer, rings the bell, and waits for input, which is terminated by any non-number. The input buffer is then restored, and the program continues. If the input was terminated by other than a period, it is ignored. If the control-S was typed during a garbage collection, the number is the new MINFS setting for the type being collected, otherwise for type 8, i.e., list words.

(MINHASH *x*) [Function]

The atom hash table automatically expands by a specified number of pages each time it fills up. The number of pages is set via the function MINHASH. The initial setting is (MINHASH 2) (room for 1024 new atoms).

(GCTRP *N*) [Function]

"Garbage Collection Trap". Causes a (simulated) control-H interrupt when the number of free list words remaining equals *N*, i.e., when a garbage collection would occur in *N* more conses. The message GCTRP is printed, the function INTERRUPT is called, and a break occurs. Note that by advising INTERRUPT the user can program the handling of a GCTRP instead of going into a break.[10]

GCTRP returns its last setting.

(GCTRP -1) will "disable" a previous GCTRP since there are never -1 free list words. GCTRP is initialized this way.

(GCTRP) returns the number of list cells left, i.e., number of CONSes until next type LISTP garbage collection.

(CLOSER *A* *X*) [Function]

Stores *X* into memory location *A*. Both *X* and *A* must be numbers.

(OPENR *A*) [Function]

Returns the number in memory location *A*, i.e., boxed.

22.8 THE ASSEMBLER AND LAP

The Interlisp-10 compiler has two principal passes. The first compiles its input into a macro assembly language called LAP.[11] The second pass expands the LAP code, producing (numerical) machine language instructions. The output of the second pass is written on a file and/or stored in binary program space.

[9] control-X for Interlisp-10 on TOPS-20.

[10] For GCTRP interrupts, INTERRUPT is called with *INTYPE* (its third argument) equal to 3. If the user does not want to go into a break, the advice should still allow INTERRUPT to be entered, but first set *INTYPE* to -1. This will cause INTERRUPT to "quietly" go away by calling the function that was interrupted. The advice should *not* exit INTERRUPT via RETURN, as in this case the function that was about to be called when the interrupt occurred would not be called.

[11] The exact form of the macro assembly language is extremely implementation dependent, as well as being influenced by the architecture and instruction set for the machine that will run the compiled program.

Input to the compiler is usually a standard Interlisp EXPR definition. However, in Interlisp-10, machine language coding can be included within a function by the use of one or more ASSEMBLE forms as described below. In other words, ASSEMBLE allows the user to write portions of a function in LAP. Note that ASSEMBLE is only a compiler directive; it has no independent definition. Therefore, functions which use ASSEMBLE must normally be compiled in order to run.[12]

22.8.1 Assemble

Note: ASSEMBLE *is provided for situations where its use is unavoidable. However, its use is definitely not encouraged. The disadvantages are several.* ASSEMBLE *code is unavoidably dependent on the PDP-10, Tenex, and implementation details of Interlisp-10. Thus,* ASSEMBLE *code is not transportable to Interlisp on another machine or operating system, and implementation changes to Interlisp-10 can (and frequently do) require changes to existing* ASSEMBLE *code.*

The format of ASSEMBLE is similar to that of PROG:

(ASSEMBLE V S_1 S_2 . . . S_N)

V is a list of variables to be bound during the first pass of the compilation, *not* during the running of the object code. The assemble statements S_1 ... S_N are compiled sequentially, each resulting in one or more instructions of object code. When run, the value of the ASSEMBLE "form" is the contents of AC1 at the end of the execution of the assemble instructions. Note that ASSEMBLE may appear anywhere in an Interlisp-10 function. For example, one may write:

```
(IGREATERP (IQUOTIENT (LOC (ASSEMBLE NIL
                                     (MOVEI 1 , -5)
                                     (JSYS 13)))
                           1000)
           4)
```

to test if job runtime exceeds 4 seconds.[13]

22.8.1.1 Assemble Statements

If an assemble statement is an atom, it is treated as a label identifying the location of the next statement that will be assembled.[14] Such labels defined in an ASSEMBLE form are like PROG labels in that they may be referenced from the current and lower level nested PROGs or ASSEMBLEs.

[12] The MACROTRAN package (page 5.19) does permit the user to run programs interpretively which contain ASSEMBLE directives. Each ASSEMBLE directive is compiled as a separate function. There is some loss in efficiency over compiling the entire function as a unit, and not all ASSEMBLE expressions are tractable to this procedure.

[13] This example is to illustrate use of ASSEMBLE, and is *not* a recommendation to use the above code. The function JSYS (page 22.6) is the appropriate method.

[14] A label can be the last thing in an ASSEMBLE form, in which case it labels the location of the first instruction *after* the ASSEMBLE form.

If an assemble statement is not an atom, CAR of the statement must be an atom and one of: (1) a number; (2) a LAP op-def (i.e., has a property value OPD); (3) an assembler macro (i.e., has a property value AMAC); or (4) one of the special assemble instructions given below, e.g., C, CQ, etc. Anything else will cause the error message OPCODE? - ASSEMBLE.

The types of assemble statements are described here in the order of priority used in the ASSEMBLE processor; that is, if an atom has both properties OPD and AMAC, the OPD will be used. Similarly a special ASSEMBLE instruction may be redefined via an AMAC. The following descriptions are of the first pass processing of ASSEMBLE statements. The second pass processing is described in the section on LAP, page 22.15.

(1) numbers

If CAR of an assemble statement is a number, the statement is not processed in the first pass (see page 22.15).

(2) LAP op-defs

The property OPD is used for two different types of op-defs: PDP-10 machine instructions, and LAP macros. If the OPD definition (i.e., the property value) is a number, the op-def is a machine instruction. When a machine instruction, e.g., HRRZ, appears as CAR of an assemble statement, the statement is not processed during the first pass but is passed to LAP. The forms and processing of machine instructions by LAP are described on page 22.16.

If the OPD definition is not a number, then the op-def is a LAP macro. When a LAP macro is encountered in an assemble statement, its arguments are evaluated and processing of the statement with evaluated arguments is left for the second pass and LAP. For example, LDV is a LAP macro, and (LDV (QUOTE X) SP) in assemble code results in (LDV X N) in the LAP code, where N is the value of SP. The form and processing of LAP macros are described on page 22.17.

(3) assemble macros

If CAR of an assemble statement has a property AMAC, the statement is an assemble macro call. There are two types of assemble macros: lambda and substitution. If CAR of the macro definition is the atom LAMBDA, the definition will be *applied* to the arguments of the call and the resulting list of statements will be assembled. For example, REPEAT could be defined as a LAMBDA macro with two arguments, N and M, which expands into N occurrences of M, e.g., (REPEAT 3 (CAR1)) expands to ((CAR1) (CAR1) (CAR1)). The definition (i.e., value of property AMAC) for REPEAT could be:

```
(LAMBDA (N M)
  (PROG (YY)
     A  (COND
          ((ILESSP N 1)
           (RETURN (CAR YY)))
          (T (SETQ YY (TCONC YY M))
             (SETQ N (SUB1 N))
             (GO A)))))
```

If CAR of the macro definition is not the atom LAMBDA, it must be a list of dummy symbols. The arguments of the macro call will be substituted for corresponding appearances of the dummy symbols in

CDR of the definition, and the resulting list of statements will be assembled.[15] For example, ABS could be a substitution macro which takes one argument, a number, and expands into instructions to place the absolute value of the number in AC1:

```
((X)
 (CQ (VAG X))
 (CAIGE 1 , 0))
 (MOVN 1 , 1))
```

(4) special assemble statements

(CQ $E_1 \cdots E_N$) CQ (compile quote) takes any number of arguments which are assumed to be regular Interlisp expressions and are compiled in the normal way. E.g.

```
(CQ (COND
      ((NULL Y)
       (SETQ Y 1)))
    (SETQ X (IPLUS Y Z)))
```

Note: to avoid confusion and minimize dependence on the current implementation, it is best to have as much of a function as possible compiled in the normal way, e.g., to load the value of X to AC1, (CQ X) is preferred to (LDV (QUOTE X) SP).

(C $E_1 \cdots E_N$) C (Compile) takes any number of arguments which are first evaluated, then compiled in the usual way. Both C and CQ permit the inclusion of regular compilation within an assemble form.

(E $E_1 \cdots E_N$) E (Evaluate) takes any number of arguments which are evaluated in sequence. For example, (PSTEP) calls a function which increments the compiler variable SP.

(SETQ *VAR*) Compiles code to set the variable *VAR* to the contents of AC1.

(VAR (*OP AC* , *VARNAME*))
Permits writing a machine instruction with the value of a variable as the operand. Generates the appropriate address and index fields to reference the value of *VARNAME*. *VARNAME* may be a locally bound variable, free variable, GLOBALVAR, etc. Note that VAR may generate more than one instruction.

(* ...) Used to indicate a comment; the statement is ignored.

22.8.1.2 COREVALs

There are several locations in the basic machine code of Interlisp-10 which may be referenced from compiled code. The current value of each location is stored on the property list under the property

[15] Note that assemble macros produce a list of statements to be assembled, whereas compiler macros produce a single expression. An assemble macro which *computes* a list of statements begins with LAMBDA and may be *either* spread or no-spread. The analogous compiler macro begins with an atom, (i.e., is always no-spread) and the LAMBDA is understood.

COREVAL.[16] Since these locations may change in different reassemblies of Interlisp-10, they are written symbolically on compiled code files, i.e., the name of the corresponding COREVAL is written, not its value. Some of the COREVALs used frequently in ASSEMBLE are:

KT	contains (pointer to) atom T
KNIL	Contains (a pointer to) the atom NIL.
MKN	Routine to box an integer.
MKFN	Routine to box floating number.
IUNBOX	Routine to unbox an integer.
FUNBOX	Routine to unbox floating number.

The index registers used for the push-down stack pointers are also included as COREVALS. These are not expected to change, and are not stored symbolically on compiled code files; however, they should be referenced symbolically in assemble code. They are:

PP	Parameter stack.
CP	Control stack.
VP	Basic frame pointer.

22.8.2 LAP

LAP (for LISP Assembly Processor) expands the output of the first pass of compilation to produce numerical machine instructions.

22.8.2.1 LAP Statements

If a LAP statement is an atom, it is treated as a label identifying the location of the next statement to be processed. If a LAP statement is not an atom, CAR of the statement must be an atom and either: (1) a number; (2) a machine instruction; or (3) a LAP macro.

(1) numbers

If CAR of a LAP statement is a number, a location containing the number is produced in the object code.[17] E.g.,

```
(ADD 1 , A (1))
     .
     .
```

[16] The value of COREVALS is a list of all atoms with COREVAL properties.

[17] Note that if a function is intended to be swappable, it may not contain any relocatable, indexed instructions.

```
A    (1)
     (4)
     (9)
```

Statements of this type are processed like machine instructions, with the initial number serving as a 36-bit op-code.

(2) Machine Instructions

If `CAR` of a LAP statement has a numeric value for the property `OPD`,[18] the statement is a machine instruction. The general form of a machine instruction is:

(*OPCODE AC* , @ *ADDRESS* (index))

OPCODE is any PDP-10 instruction mnemonic or Interlisp UUO.[19]

AC, the accumulator field, is optional. However, if present, it *must* be followed by a comma. *AC* is either a number or an atom with a `COREVAL` property. The low order 4 bits of the number or `COREVAL` are OR'd to the AC field of the instruction.

@ may be used anywhere in the instruction to specify indirect addressing (bit 13 set in the instruction) e.g., `(HRRZ 1 , @ 1 (VP))`.

ADDRESS is the address field which may be any of the following:

= *CONSTANT*
Reference to an unboxed constant. A location containing the unboxed constant will be created in a region at the end of the function, and the address of the location containing the constant is placed in the address field of the current instruction. The constant may be a number e.g., `(CAME 1 , = 3596)`; an atom with a property `COREVAL` (in which case the constant is the value of the property, at `LOAD` time); any other atom which is treated as a label (the constant is then the address of the labeled location) e.g., `(MOVE 1 , = TABLE)` is equivalent to `(MOVEI 1 , TABLE)`; or an expression whose value is a number.

' *POINTER*
The address is a reference to a Interlisp pointer, e.g., a list, number, string, etc. A location containing the pointer is assembled at the end of the function, and the current instruction will have the address of this location, e.g.,

```
(HRRZ 1 , ' "IS NOT DEFINED")

(HRRZ 1 , ' (NOT FOUND))
```

*
Specifies the current location in the compiled function; e.g., `(JRST * 2)` has the same effect as `(SKIPA)`.

a literal atom
If the atom has a property `COREVAL`, it is a reference to a system location, e.g., `(SKIPA 1 , KNIL)`, and the address used is the value of the `COREVAL`.

[18] The value is an 18 bit quantity (rather than 9), since some UUO's also use the AC field of the instruction.

[19] The TENEX JSYS's are not defined, that is, one must write `(JSYS 107)` instead of `(KFORK)`.

Otherwise the atom is a label referencing a location in the LAP code, e.g., (JRST A).

a number — The number is the address; e.g.,

```
(MOVSI 1 , 400000Q)
(HLRZ 2 , 1 (1))
```

a list — The form is evaluated, and its value is the address.

Anything else in the address field causes an error message, e.g., (SKIPA 1 , KNILL) - LAPERROR. A number may follow the address field and will be added to it, e.g., (JRST A 2).

INDEX is denoted by a *list* following the address field, i.e., the address field *must* be present if an index field is to be used. The index (CAR of the list) must be either a number, or an atom with a property COREVAL, e.g., (HRRZ 1 , 0 (1)).

(3) LAP macros

If CAR of a LAP statement is the name of a LAP macro, i.e., has the property OPD, the statement is a macro call. The arguments of the call follow the macro name: e.g., (LQ2 FIE 3).

LAP macro calls comprise most of the output of the first pass of the compiler, and may also be used in ASSEMBLE. The definitions of these macros are stored on the property list under the property OPD, and like assembler macros, may be either lambda or substitution macros. In the first case, the macro definition is applied to the arguments of the call;[20] in the second case, the arguments of the call are substituted for occurrences of the dummy symbols in the definition. In both cases, the resulting list of statements is again processed, with macro expansion continuing till the level of machine instructions is reached.

Some examples of LAP macros are shown below.

```
(DEFLIST
'[(LQ ((X)                               (* LOAD QUOTE TO AC1)
      (HRRZ 1 , ' X)))
  (LQ2 ((X AC)                           (* LOAD QUOTE TO AC)
        (HRRZ AC , ' X)))
  (LDV ((A SP)                           (* LOAD LOCAL VARIABLE TO AC1)
        (HRRZ 1 , (VREF A SP))))
  (STV ((A SP)                           (* SET LOCAL VARIABLE FROM AC1)
        (HRRM 1 , (VREF A SP))))
  (LDV2 ((A SP AC)                       (* LOAD LOCAL VARIABLE TO AC)
         (HRRZ AC , (VREF A SP))))
  (LDF ((A SP)                           (* LOAD FREE VARIABLE TO AC1)
        (HRRZ 1 , (FREF A SP))))
  (STF ((A SP)                           (* SET FREE VARIABLE FROM AC1)
        (HRRM 1 , (FREF A SP))))
  (LDF2 ((A SP)                          (* LOAD FREE VARIABLE TO AC)
         (HRRZ 2 , (FREF A SP))))
  (CAR1 (NIL                             (* CAR OF AC1 TO AC1)
```

[20] The arguments were already evaluated in the first pass, see page 22.13.

```
        (HRRZ 1 , 0 (1))))
  (CDR1 (NIL                              (* CDR OF AC1 TO AC1)
        (HLRZ 1 , 0 (1))))
  (CAR2 ((AC)                             (* CAR OF AC TO AC)
        (HRRZ AC , 0 (AC))))
  (CLL ((NAM N)                           (* CALL FN WITH N ARGS GIVEN)
       (CCALL N , ' NAM)))
  (LCLL ((NAM N)                          (* LINKED CALL WITH N ARGS)
        (LNCALL N , (MKLCL NAM))))
  (RET (NIL                               (* RETURN FROM FN)
      (POPJ CP ,))
  (PUSHP (NIL (PUSH PP , 1)))
  (PUSHQ ((X)                             (* PUSH QUOTE)
         (PUSH PP , ' X)))]
'OPD)
```

22.8.3 Using Assemble

In order to use ASSEMBLE, it is helpful to know the following things about how compiled code is run. All variable bindings and temporary pointers are stored on the parameter pushdown stack (addressed by index register PP). Control information is stored on the control pushdown stack (addressed by index register CP). A function call proceeds as follows:

1. The calling function pushes the argument values on the parameter stack.

2. The calling function invokes a routine that adjusts the number of arguments if too few or too many were supplied, and binds the arguments. Binding usually implies the creation of a basic frame.[21]

3. Then the called function is run.

The arguments in the basic frame are referenced relative to index register VP, e.g., 1(VP) addresses the first argument. However, it is better to reference variables in less implementation dependent ways, such as (CQ ...) or (VAR (...)). The compiler will then generate the correct code whether the variable is bound locally, is a free reference, is a GLOBALVAR, etc.

The parameter stack may be used for temporary storage of pointers. Both halves of a word on the parameter stack may be pointers. On the control stack the right half of a word must be a pointer, the left a non-pointer. Anything else can cause the garbage collector to fail.

For temporary storage of unboxed numbers, the following ASSEMBLE macros are provided:

(PUSHN *ADDR*) "Pushes" the number referenced by *ADDR*. *ADDR* may be any legal ASSEMBLE code address field, for example: (PUSHN 1), (PUSHN = 0), (PUSHN @ 2)

(POPN *ADDR*) "Pops" the most recent number to *ADDR*.

[21] Whether a basic frame is created for a PROG or open lambda depends on whether any of the variables are specvars.

(NREF (*OP AC* , *N*))
References a previously pushed number. *OP* is the opcode, *AC* is the accumulator, *N* is the relative position of the desired number on the pseudo number stack. That is, *N* = 0 refers to the most recent number, *N* = -1 to the next most recent, etc. For example: (NREF (MOVN 1, -1))

(PUSHNN $N_1 \cdots N_M$)
"Pushes" a sequence of numbers specified by N_i where N_i is a list of any legal address field. For example: (PUSHNN (1) (2) (= 0)) pushes the contents of AC1, the contents of AC2, and the constant 0.

(POPNN *N*) "Pops" the *N* most recent numbers, discarding the values.

Use of these macros is subject to the following restrictions:

1. PUSHN's and POPN's must be seen by the compiler in the same order and number in which they are executed. The compiler does not analyze the code; it assumes when it encounters a PUSHN in the sequential processing of the code that the PUSHN will in fact be executed.

2. Every number that is pushed must be popped.

3. In nested ASSEMBLE statements, if a PROG or open lambda occurs between the inner and outer level ASSEMBLE, numbers pushed in the outer ASSEMBLE may not be referenced from the inner ASSEMBLE.

The value of a function is always returned in AC1. Therefore, the pseudo-function, AC, is available for obtaining the current contents of AC1. For example (CQ (FOO (AC))) compiles a call to FOO with the current contents of AC1 as argument, and is equivalent to:

```
(PUSHP)
(E (PSTEP))
(CLL (QUOTE FOO) 1)
(E (PSTEPN -1))
```

In using AC, be sure that it appears as the first argument to be evaluated in the expression. For example: (CQ (IPLUS (LOC (AC)) 2))

There are several ways to reference the values of variables in assemble code. For example:

(CQ X) Puts the value of X in AC1.

(LDV2 (QUOTE X) SP 3)
Puts the value of X in AC3.

(SETQ X) Sets X to the contents of AC1.

(VAR (HRRM 2 , X))
Sets X to the contents of AC2.

(CQ (LOC (AC)))
Boxes the contents of AC1.

(FASTCALL MKFN)
Floating boxes the contents of AC1.

`(CQ (VAG X))` Puts the unboxed value of `X` in AC1.

`(FASTCALL FUNBOX)`
Gets the floating unbox of AC1.

To call a function directly, the arguments must be pushed on the parameter stack, and `SP` must be updated, and then the function called: e.g.,

```
(CQ (CAR X))
(PUSHP)                  (* stack first argument)
(E (PSTEP))
(PUSHQ 3.14)
(E (PSTEP))              (* stack second argument)
(CLL (QUOTE FUM) 2)      (* call FUM with 2 arguments)
(E (PSTEPN -2))          (* adjust stack count)
```

and is equivalent to:

```
(CQ (FUM (CAR X) 3.14))
```

22.9 INTERFORK COMMUNICATION IN INTERLISP-10

The functions described below permit two forks (one or both of them Interlisp-10) to have a common area of address space for communication by providing a means of assigning a block of storage *guaranteed not to move during garbage collections.*

`(GETBLK` *N*`)` [Function]

Creates a block *N pages* in size (512 words per page). Value is the address of the first word in the block, which is a multiple of 512 since the block will always begin at a page boundary. If not enough pages are available, generates the error `ILLEGAL OR IMPOSSIBLE BLOCK`.

Note: the block can be used for storing unboxed numbers ONLY.

To store a number in the block, the following function could be defined:

```
(SETBLOCK (LAMBDA (START N X) (CLOSER (IPLUS (LOC START) N) X]
```

Some boxing and unboxing can be avoided by making this function compile open via a substitution macro.

Note: `GETBLK` *should be used sparingly since several unmovable regions of memory can make it difficult or impossible for the garbage collector to find a contiguous region large enough for expanding array space.*

`(RELBLK` *ADDRESS N*`)` [Function]

releases a block of storage beginning at *ADDRESS* and extending for *N* pages. Causes an error `ILLEGAL OR IMPOSSIBLE BLOCK` if any of the range is not a block. Value is *ADDRESS*.

22.10 SUBSYS

This section describes a function, SUBSYS, which permits the user to run a Tenex/TOPS-20 subsystem, such as SNDMSG, SRCCOM, TECO, or even another Interlisp, from inside of an Interlisp without destroying the latter. In particular, (SUBSYS 'EXEC) will start up a lower exec, which will print the operating system herald, followed by @. The user can then do anything at this exec level that he can at the top level, without affecting his superior Interlisp. For example, he can start another Interlisp, perform a SYSIN, run for a while, type a control-C returning him to the lower exec, RESET, do a SNDMSG, etc. The user exits from the lower exec via the command QUIT,[22] which will return control to SUBSYS in the higher Interlisp. Thus with SUBSYS, the user need not perform a SYSOUT to save the state of his Interlisp in order to use a Tenex/TOPS-20 capability which would otherwise clobber the core image. Similarly, SUBSYS provides a way of checking out a SYSOUT file in a fresh Interlisp without having to commandeer another terminal or detach a job.

While SUBSYS can be used to run any subsystem directly, without going through an intervening exec, this procedure is not recommended. The problem is that control-C always returns control to the next highest *EXEC*. Thus if the user is running an Interlisp in which he performs (SUBSYS 'LISP), and then types control-C to the lower Interlisp, control will be returned to the exec above the first Interlisp. If the user elects to call a subsystem directly, he must therefore know how it is normally exited and always exit from it that way.[23]

Starting a lower exec does not have this disadvantage, since it can *only* be exited via QUIT or POP, i.e., the lower exec is effectively "errorset protected" against control-C.

(SUBSYS *FILE/FORK INCOMFILE OUTCOMFILE ENTRYPOINTFLG*) [Function]

If *FILE/FORK*=EXEC, starts up a lower exec, otherwise runs <SUBSYS>system, e.g. (SUBSYS 'SNDMSG), (SUBSYS 'TECO) etc. (SUBSYS) is the same as (SUBSYS 'EXEC). Control-C always returns control to next higher *EXEC*. Note that more than one Interlisp can be stacked, but there is no backtrace to help you figure out where you are.

INCOMFILE and *OUTCOMFILE* provide a way of specifying files for input and output. *INCOMFILE* can also be a string, in which case a temporary file is created, and the string printed on it.

ENTRYPOINTFLG may be START, REENTER, or CONTINUE. NIL is equivalent to START, except when *FILE/FORK* is a handle (see below) in which case NIL is equivalent to CONTINUE.

The value of SUBSYS is a large integer which is a handle to the lower fork. The lower fork is *not* reset unless the user specifically does so using KFORK, described below.[24] If SUBSYS is given as its first

[22] POP on TOPS-20.

[23] Interlisp is exited via the function LOGOUT, TECO via the command ;H, SNDMSG via control-Z, and EXEC via QUIT.

[24] The fork is also reset when the handle is no longer accessible, i.e., when nothing in the Interlisp system points to it. Note that the fork is accessible while the handle remains on the history list.

argument the value of a previous call to SUBSYS,[25] it continues the subsystem run by that call. For example, the user can do (SETQ SOURCES (SUBSYS 'TECO)), load up the TECO with a big source file, massage the file, leave TECO with ;H, run Interlisp for awhile (possibly including other calls to SUBSYS) and then perform (SUBSYS 'SOURCES) to return to TECO, where he will find his file loaded and even the TECO pointer position preserved.

Note that if the user starts a lower EXEC, in which he runs an Interlisp, control-C's from the Interlisp, then QUIT from the EXEC, if he subsequently continues this EXEC with SUBSYS, he can reenter or continue the Interlisp.

Note also that calls to SUBSYS can be stacked. For example, using SUBSYS, the user can run a lower Interlisp, and within that Interlisp, yet another, etc., and ascend the chain of Interlisps using LOGOUT, and then descend back down again using SUBSYS.

For convenience, (SUBSYS T) continues the last subsystem run.

SNDMSG, LISP, TECO, and EXEC are all LISPXMACROS (page 8.19) which perform the corresponding calls to SUBSYS. CONTIN is a LISPXMACRO which performs (SUBSYS T), thereby continuing the last SUBSYS.[26]

(KFORK *FORK*) [Function]

Accepts a value from SUBSYS and kills it (RESET in Tenex terminology). If (SUBSYS *FORK*) is subsequently performed, an error is generated. (KFORK T) kills all outstanding forks (from this Interlisp).

22.11 JFN FUNCTIONS IN INTERLISP-10

JFN stands for Job File Number. It is an integral part of the Tenex file system and is described in [Murl], and in somewhat more detail in the Tenex JSYS manual. In Interlisp-10, the following functions are available for direct manipulation of JFNs:

(OPNJFN *FILE ACCESS*) [Function]

Returns the JFN for *FILE*. If *FILE* not open, generates a FILE NOT OPEN error. *ACCESS*=NIL, INPUT, OUTPUT, or BOTH as described in discussion of OPENP. For example, (JSYS 51Q (OPNJFN FILE) BYTE) will write a byte on a file, while (JSYS 50Q (OPNJFN FILE) NIL NIL 2) will read one byte.

(GTJFN *FILE EXT V FLAGS*) [Function]

Sets up a "long" call to GTJFN (see JSYS manual). *FILE* is a file name possibly containing control-F and/or <esc>. *EXT* is the default extension, *V* the default version (overriden if *FILE* specifies extension/version, e.g., FOO.COM;2). *FLAGS* is

[25] Must be the exact same large number, i.e., EQ. Note that if the user neglects to set a variable to the value of a call to SUBSYS, (and has performed an intervening call so that (SUBSYS T) will not work), he can still continue this subsystem by obtaining the value of the call to SUBSYS for the history list using the function VALUEOF, described in page 8.16.

[26] The EXEC LISPXMACRO is defined to save its value on LASTEXEC so that subsequent EXEC commands will restart the same exec.

as described on page 17, section 2 of JSYS manual. *FILE* and *EXT* may be strings or atoms; *V* and *FLAGS* must be numbers. Value is JFN, or NIL on errors.

(RLJFN *JFN*) [Function]

Releases *JFN*. (RLJFN -1) releases all JFN's which do not specify open files. Value of RLJFN is T.

(JFNS *JFN AC3 STRPTR*) [Function]

Converts *JFN* (a small number) to a file name. *AC3* is either NIL, meaning format the file name as would OPENP or other Interlisp-10 file functions, or else is a number, meaning format according to JSYS manual. The value of JFNS is atomic except where enough options are specified by *AC3* to exceed atom size. In this case, the value is returned as a string.

STRPTR is an optional string pointer to be reused. In this case, the string characters are stored in an internal scratch string, MACSCRATCHSTRING, so that a subsequent call to JFNS will overwrite the characters returned by this one. The value of JFNS when *STRPTR* is supplied is always a string.

The following function is available in Interlisp-10 for specialized file applications:

(OPENF *FILE X*) [Function]

Opens *FILE*. *X* is a number whose bits specify the access and mode for *FILE*, i.e., *X* corresponds to the second argument to the Tenex JSYS OPENF (see JSYS Manual). Value is full name of *FILE*.

The first argument to OPENF can also be a number, which is then interpreted as a JFN. OPENF does not affect the primary input or output file settings, and does not check whether the file is already open - i.e., the same file can be opened more than once, possibly for different purposes.

Note that for almost all applications the function OPENFILE (page 6.1) provides a more convenient (and implementation independent) way of opening files.

22.12 DISPLAY TERMINALS

The value of the variable DISPLAYTERMFLG indicates whether the user is running on a display terminal or not. DISPLAYTERMFLG is used in various places in the system, e.g., PRETTYPRINT, HELPSYS, etc., primarily to decide how much information to present to the user (more on a display terminal than on a hard copy terminal). DISPLAYTERMFLG is initialized to the value of (DISPLAYTERMP), whenever Interlisp is (re)-entered, and after returning from a sysout.

(DISPLAYTERMP) [Function]

Value is T if user is on a display terminal, NIL otherwise. In Interlisp-10, DISPLAYTERMP is defined to invoke the appropriate jsys to check the user's terminal type.

22.13 THE INTERLISP-10 SWAPPER

Interlisp-10 provides a very large auxilary address space exclusively for swappable arrays (primarily compiled function definitions). In addition to the 256K of *resident* address space, this "shadow space" can currently accomodate an additonal 256K words, can easily be expanded to 3.5 million words, and with some further modifications, could be expanded to 128 million words. Thus, the overlay system provides essentially unlimited space for compiled code.[27]

Shadow space and the swapper are intended to be more or less transparent to the user. However, this section is included in the manual to give programmers a reasonable feeling for what overlays are like, without getting unnecessarily technical, as well as to document some new functions and system controls which may be of interest for authors of exceptionally large systems.

22.13.1 Overlays

The shadow space is a very large auxiliary address space used exclusively for an Interlisp data-type called a swappable array. The regular address space is called the "resident" space to distinguish it from shadow space. Any kind of resident array - compiled code, pointer data, binary data, or a hash array - can be copied into shadow space ("made swappable"), from which it is referred to by a one-word resident entity called a handle. The resident space occupied by the original array can then be garbage collected normally (assuming there are no remaining pointers to it, and it has not been made shared by a MAKESYS). Similarly, a swappable array can be made resident again at any time, but of course this requires (re)allocating the necessary resident space.

The main purpose and intent of the swapping system is to permit utilization of swappable arrays directly and interchangeably with resident arrays, thereby saving resident space which is then available for other data-types, such as lists, atoms, strings, etc.

This is accomplished as follows: A section of the resident address space is permanently reserved for a *swapping buffer.*[28] When a particular swappable array is requested, it is brought (swapped) in by mapping or *overlaying* the pages of shadow space in which it lies onto a section of the swapping buffer. This process is the swapping or overlaying from which the system takes its name. The array is now (directly) accessible. However, further requests for swapping could cause the array to be overlaid with something else, so in effect it is liable to go away at any time. Thus all system code that relates to arrays must recognize handles as a special kind of array, fetch them into the buffer (if not already there), when necessary check that they have not disappeared, fetch them back in if they have, and even be prepared for the second fetch to bring the swappable array in at a different place than did the first.

The major emphasis in the design of the overlay system has been placed on running compiled code, because this accounts for the overwhelming majority of arrays in typical systems, and for as much as 60% of the overall data and code. The system supports the running of compiled code directly from the

[27] Since compiled code arrays point to atoms for function names, and strings for error messages, not to mention the fact that programs usually have data base, which are typically lists rather than arrays, there is still a very real and finite limit to the total size of programs that Interlisp-10 can accomodate. However, since much of the system and user compiled code can be made swappable, there is that much more resident space available for these other data-types.

[28] Initially 64,512 word pages, but can be changed via the function SETSBSIZE described below.

swapping buffer, and the function calling mechanism knows when a swappable definition is being called, finds it in the buffer if it is already there, and brings it in otherwise. Thus, from the user's point of view, there is no need to distinguish between swappable and resident compiled definitions, and in fact CCODEP will be true for either.

22.13.2 Efficiency

Once of the most important design goals for the overlay system was that swappable code should not execute any extra instructions compared to resident code, once it had been swapped in. Thus, the instructions of a swappable piece of code are identical (except for two instructions at the entry point) to those of the resident code from which it was copied,[29] and similarly when a swappable function calls another function (of any kind) it uses the exact same calling sequence as any other code. Thus, all costs associated with running of swappable code are paid at the point of entry (both calling and returning).[30]

The cost of the swapping itself, i.e. the fetch of a new piece of swapped code into the buffer, is even harder to measure meaningfully, since two successive fetches of the same function are not the same, due to the fact that the instance created by the first fetch is almost certain to be resident when the second is done, if no swapping is done in between. Similarly, two successive PMAP's (the Tenex operation to fetch one page) are not the same from one moment to another, even if the virtual state of both forks is exactly the same - a difficult constraint to meet in itself.[31] Thus, all that can be reported is that empirical measurements and observations have shown no consistent slowdown in performance of systems containing swappable functions viz a viz resident functions.

22.13.3 Specifications

Associated with the overlay system is a datatype called a SWPARRAY, (type name SWPARRAYP), which occupies one word of resident space, plus however much of shadow space needed for the body of the array. ARGLIST, FNTYP, NARGS, GETD, PUTD, ARGTYPE, ARRAYSIZE, CHANGENAME, CALLS, BREAK, ADVISE, and EDITA all work equally well with swappable as resident programs. CCODEP is true for all compiled functions/definitions.

(SWPARRAYP *x*) [Function]

Analogous to ARRAYP. Returns *x* if *x* is a swappable array and, NIL otherwise.

[29] The relocatable instructions are indexed by a base register, to make them run equally well at any location in the buffer. The net slowdown due to this extra level of indirection is too small to measure accurately in the overall running of a program. On analytical grounds, one would expect it to be around 2%.

[30] If the function in question does nothing, e.g. a compiled (LAMBDA NIL NIL), it costs approximately twice as much to enter its definition if it is swappable as compared to resident. However, very small functions are normally not made swappable (see MKSWAPP, page 22.26), because they don't save much space, and are (typically) entered frequently. Larger programs don't exhibit a measurable slow down since they amortize the entry cost over longer runs.

[31] The cost of fetching is probably not in the mapping operation itself but in the first reference to the page, which has a high probability of faulting. This raises the problem of measuring page fault activity, another morass of uncertainty.

(SCODEP *x*) [Function]

Analogous to CCODEP. Returns T if *x* is or has a swapped compiled definition.

(MKSWAP *x*) [Function]

If *x* is a resident array, returns a swappable array which is a copy of *x*. If *x* is a literal atom and (CCODEP *x*) is true, its definition is copied into a swappable array, and it is (undoably) redefined with the latter. MKSWAP returns *x*.

(MKUNSWAP *x*) [Function]

The inverse of MKSWAP. *x* is either a swappable array, or an atom with swapped definition on its CODE property.

(MKSWAPP *FNAME CDEF*) [Function]

All compiled definitions begin life as resident arrays, whether they are created by LOAD, or by compiling to core. Before they are stored away into their atom's function cell, MKSWAPP is applied to the atom and the array. If the value of MKSWAPP is T, the definition is made swappable; otherwise, it is left resident. By redefining MKSWAPP or advising it, the user can completely control the swappability of all future definitions as they are created. The initial definition of MKSWAPP will make a function swappable if (1) NOSWAPFLG is NIL, and (2) the name of the function is not on NOSWAPFNS, and (3) the size of its definition is greater than MKSWAPSIZE words, initially 128.

(SETSBSIZE *N*) [Function]

Sets the size of the swapping buffer to *N*, a number of *pages*. Returns the previous value. (SETSBSIZE) returns the current size without changing it.

Note: Currently, the system lacks error recovery routines for situations such as a call to a swappable function which is too big for the swapping buffer, or when the size is zero. Therefore, SETSBSIZE should be used with care.

CHAPTER 23

LISPUSERS PACKAGES

This chapter describes packages which are of sufficient utility that they would otherwise be included as part of the Interlisp system, except for virtual address space limitations. These packages normally reside on the directory <LISPUSERS>.

23.1 PATTERN MATCH COMPILER

Note: The pattern match compiler is a LispUsers package which can be loaded from the file MATCH.DCOM. *The entries have a* FILEDEF *property (see page 15.8), so simply using a pattern match construct will cause the file to be loaded automatically.*

The pattern match compiler provides a fairly general pattern match facility within CLISP. This facility allows the user to specify certain tests that would otherwise be clumsy to write, by giving a pattern which the datum is supposed to match. Essentially, the user writes "Does the (expression) X look like (the pattern) P?" For example, `X:(& 'A -- 'B)` asks whether the second element of `X` is an `A`, and the last element a `B`. The implementation of the matching is performed by computing (once) the equivalent Interlisp expression which will perform the indicated operation, and substituting this for the pattern, and *not* by invoking each time a general purpose capability such as that found in FLIP or PLANNER. For example, the translation of `X:(& 'A -- 'B)` is:

```
(AND (EQ (CADR X) 'A)
     (EQ (CAR (LAST X)) 'B))
```

Thus the CLISP pattern match facility is really a Pattern Compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g., in inner loops. As a result, the CLISP pattern match facility does not contain (yet) some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, the user can be confident that what facilities it does provide will result in Interlisp expressions comparable to those he would generate by hand.[1]

The syntax for pattern match expressions is *FORM*:*PATTERN*, where *PATTERN* is a list as described below. As with iterative statements, the translation of patterns, i.e., the corresponding Interlisp expressions, are stored in the hash array CLISPARRAY (see page 16.19). The original expression, *FORM*:*PATTERN*, is replaced by an expression of the form (MATCH *FORM* WITH *PATTERN*). CLISP also recognizes expressions input in this form.

[1] Wherever possible, already existing Interlisp functions are used in the translation, e.g., the translation of `($ 'A $)` uses MEMB, `($ ('A $) $)` uses ASSOC, etc.

If *FORM* appears more than once in the translation, and it is not either a variable, or an expression that is easy to (re)compute, such as (CAR Y), (CDDR Z), etc., a dummy variable will be generated and bound to the value of *FORM* so that *FORM* is not evaluated a multiple number of times. For example, the translation of (FOO X):($ 'A $) is simply (MEMB 'A (FOO X)), while the translation of (FOO X):('A 'B --) is:

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CAR (SETQ $$2 (FOO X)))
               'A)
           (EQ (CADR $$2) 'B]
```

In the interests of efficiency, the pattern match compiler assumes that all lists end in NIL, i.e., there are no LISTP checks inserted in the translation to check tails. For example, the translation of X:('A & --) is (AND (EQ (CAR X) (QUOTE A)) (CDR X)), which will match with (A B) as well as (A . B). Similarly, the pattern match compiler does not insert LISTP checks on elements, e.g., X:(('A --) --) translates simply as (EQ (CAAR X) 'A), and X:(($1 $1 --) --) as (CDAR X).[2] Note that the user can explicitly insert LISTP checks himself by using @, as described below, e.g., X:(($1 $1 --)@LISTP --) translates as (CDR (LISTP (CAR X))).

23.1.1 Pattern Elements

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment. (cf. the editor's pattern matcher, "--" matches any arbitrary segment of a list, while & or a subpattern match only one element of a list.) Those patterns which may match a segment of a list are called *segment* patterns; those that match a single element are called *element* patterns.

23.1.2 Element Patterns

There are several types of element patterns, best given by their syntax:

$1 or & — Matches an arbitrary element of a list.

'*EXPRESSION* — Matches only an element which is equal to the given expression e.g., 'A, '(A B).

EQ, MEMB, and ASSOC are automatically used in the translation when the quoted expression is atomic, otherwise EQUAL, MEMBER, and SASSOC.

=*FORM* — Matches only an element which is EQUAL to the value of *FORM*, e.g., =X, =(REVERSE Y).

==*FORM* — Same as =, but uses an EQ check instead of EQUAL.

[2] The insertion of LISTP checks for *elements* is controlled by the variable PATLISTPCHECK. When PATLISTPCHECK is T, LISTP checks are inserted, e.g., X:(('A --) --) translates as: (EQ (CAR (LISTP (CAR (LISTP X)))) 'A). PATLISTPCHECK is initially NIL. Its value can be changed within a particular function by using a local CLISP declaration (see page 16.10).

ATOM

The treatment depends on setting of `PATVARDEFAULT`. If `PATVARDEFAULT` is `'` or `QUOTE`, same as `'`*ATOM*. If `PATVARDEFAULT` is `=` or `EQUAL`, same as `=`*ATOM*. If `PATVARDEFAULT` is `==` or `EQ`, same as `==`*ATOM*. If `PATVARDEFAULT` is `←` or `SETQ`, same as *ATOM*`←&`. `PATVARDEFAULT` is initially `'`.

`PATVARDEFAULT` can be changed within a particular function by using a local CLISP declaration (see page 16.10).

Note: numbers and strings are always interpreted as though `PATVARDEFAULT` were `=`, regardless of its setting. `EQ`, `MEMB`, and `ASSOC` are used for comparisons involving small integers.

(*PATTERN*$_1$ ··· *PATTERN*$_N$) $N \geq 1$

Matches a list which matches the given patterns, e.g., `(& &)`, `(-- 'A)`.

ELEMENT-PATTERN`@`*FN*

Matches an element if *ELEMENT-PATTERN* matches it, and *FN* (name of a function or a `LAMBDA` expression) applied to that element returns non-`NIL`. For example, `&@NUMBERP` matches a number and `('A --)@FOO` matches a list whose first element is `A`, and for which `FOO` applied to that list is non-`NIL`.

For "simple" tests, the function-object is applied before a match is attempted with the pattern, e.g., `((-- 'A --)@LISTP --)` translates as `(AND (LISTP (CAR X)) (MEMB 'A (CAR X)))`, not the other way around. *FN* may also be a *FORM* in terms of the variable `@`, e.g., `&@(EQ @ 3)` is equivalent to `=3`.

`*`

Matches any arbitrary element. If the entire match succeeds, the element which matched the `*` will be returned as the value of the match.

Note: Normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-`NIL` if the match succeeds and `NIL` if it fails. However, if a `*` appears in the pattern, the expression generated could also return `NIL` if the match succeeds and `*` was matched to `NIL`. For example, `X:('A * --)` translates as `(AND (EQ (CAR X) 'A) (CADR X))`, so if `X` is equal to `(A NIL B)` then `X:('A * --)` returns `NIL` even though the match succeeded.

`~`*ELEMENT-PATTERN*

Matches an element if the element is *not* matched by *ELEMENT-PATTERN*, e.g., `~'A`, `~=X`, `~(-- 'A --)`.

`(*ANY*` *ELEMENT-PATTERN* *ELEMENT-PATTERN* ···`)`

Matches if any of the contained patterns match.

23.1.3 Segment Patterns

`$` or `--`

Matches any segment of a list (including one of zero length).

The difference between `$` and `--` is in the type of search they generate. For example, `X:($ 'A 'B $)` translates as `(EQ (CADR (MEMB 'A X)) 'B)`, whereas `X:(-- 'A 'B $)` translates as:

```
[SOME X
```

```
(FUNCTION (LAMBDA ($$2 $$1)
              (AND (EQ $$2 'A)
                   (EQ (CADR $$1) 'B]
```

Thus, a paraphrase of ($ 'A 'B $) would be "Is the element following the *first* A a B?", whereas a paraphrase of (-- 'A 'B $) would be "Is there *any* A immediately followed by a B?" Note that the pattern employing $ will result in a more efficient search than that employing --. However, ($ 'A 'B $) will not match with (X Y Z A M O A B C), but (-- 'A 'B $) will.

Essentially, once a pattern following a $ matches, the $ never resumes searching, whereas -- produces a translation that will always continue searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both -- and $ will be the same. For example, both X:($ 'A $3 $) and (-- 'A $3 --) translate as (CDDDR (MEMB (QUOTE A) X)), because if there are not three elements following the first A, there certainly will not be three elements following subsequent A's, so there is no reason to continue searching, even for --. Similarly, ($ 'A $ 'B $) and (-- 'A -- 'B --) are equivalent.

$2, $3, etc. Matches a segment of the given length. Note that $1 is not a segment pattern.

!*ELEMENT-PATTERN*

Matches any segment which *ELEMENT-PATTERN* would match as a list. For example, if the value of FOO is (A B C), !=FOO will match the segment ··· A B C ··· etc. Note that !* is permissible and means *VALUE-OF-MATCH*←$, e.g., X:($ 'A !*) translates to (CDR (MEMB 'A X)).

Note: since ! appearing in front of the last pattern specifies a match with some *tail* of the given expression, it also makes sense in this case for a ! to appear in front of a pattern that can only match with an atom, e.g., ($2 !'A) means match if CDDR of the expression is the atom A. Similarly, X:($!'A) translates to (EQ (CDR (LAST X)) 'A).

!*ATOM* treatment depends on setting of PATVARDEFAULT. If PATVARDEFAULT is ' or QUOTE, same as !'*ATOM* (see above discussion). If PATVARDEFAULT is = or EQUAL, same as !=*ATOM*. If PATVARDEFAULT is == or EQ, same as !==*ATOM*. If PATVARDEFAULT is ← or SETQ, same as *ATOM*←$.

. The atom "." is treated *exactly* like "!". In addition, if a pattern ends in an atom, the "." is first changed to "!", e.g., ($1 . A) and ($1 ! A) are equivalent, even though the atom "." does not explicitly appear in the pattern.

One exception where "." is not treated like "!": "." preceding an assignment does not have the special interpretation that "!" has preceding an assignment (see below). For example, X:('A . FOO←'B) translates as:

```
(AND (EQ (CAR X) 'A)
     (EQ (CDR X) 'B)
     (SETQ FOO (CDR X)))
```

but X:('A ! FOO←'B) translates as:

```
(AND (EQ (CAR X) 'A)
     (NULL (CDDR X))
```

```
                    (EQ (CADR X) 'B)
                    (SETQ FOO (CDR X)))
```

SEGMENT-PATTERN@FUNCTION-OBJECT

Matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-NIL. For example, ($@CDDR 'D $) matches (A B C D E) but not (A B D E), since CDDR of (A B) is NIL.

Note: an @ pattern applied to a segment will require *computing* the corresponding structure (with LDIFF) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

23.1.4 Assignments

Any pattern element may be preceded by "*VARIABLE*←", meaning that if the match succeeds (i.e., everything matches), *VARIABLE* is to be set to the thing that matches that pattern element. For example, if X is (A B C D E), X:($2 Y←$3) will set Y to (C D E). Note that assignments are not performed until the entire match has succeeded, so assignments cannot be used to specify a search for an element found earlier in the match, e.g., X:(Y←$1 =Y --)[3] will *not* match with (A A B C ...), unless, of course, the value of Y was A before the match started. This type of match is achieved by using place-markers, described below.

If the variable is preceded by a !, the assignment is to the *tail* of the list as of that point in the pattern, i.e., that portion of the list matched by the remainder of the pattern. For example, if X is (A B C D E), X:($!Y←'C 'D $) sets Y to (C D E), i.e., CDDR of X. In other words, when ! precedes an assignment, it acts as a modifier to the ←, and has no effect whatsoever on the pattern itself, e.g., X:('A 'B) and X:('A !FOO←'B) match identically, and in the latter case, FOO will be set to CDR of X.

Note: *←*PATTERN-ELEMENT* and !*←*PATTERN-ELEMENT* are acceptable, e.g., X:($ 'A *←('B --) --) translates as:

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CAADR (SETQ $$2 (MEMB 'A X)))  'B)
           (CADR $$2]
```

23.1.5 Place-Markers

Variables of the form #*N*, *N* a number, are called place-markers, and are interpreted specially by the pattern match compiler. Place-markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e., they can be assigned values, or used freely in forms appearing in the pattern, e.g., X:(#1←$1 =(ADD1 #1)) will match the list (2 3). However, they are not really variables in the sense that they are not bound, nor can a function called

[3] The translation of this pattern is: (COND ((AND (CDR X) (EQUAL (CADR X) Y)) (SETQ Y (CAR X)) T)). The AND is used because if Y is NIL, the pattern should match with (A NIL), but not with just (A). The T is because (CAR X) might be NIL.

from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of PATVARDEFAULT, the first appearance of a defaulted place-marker is interpreted as though PATVARDEFAULT were ←. Thus the above pattern could have been written as X:(1 =(ADD1 1)). Subsequent appearances of a place-marker are interpreted as though PATVARDEFAULT were =. For example, X:(#1 #1 --) is equivalent to X:(#1←$1 =#1 --), and translates as (AND (CDR X) (EQUAL (CAR X) (CADR X)). (Note that (EQUAL (CAR X) (CADR X)) would incorrectly match with (NIL).)

23.1.6 Replacements

The construct *PATTERN-ELEMENT←FORM* specifies that if the match succeeds, the part of the data that matched is to be *replaced* with the value of *FORM*. For example, if X =(A B C D E), X:($ 'C $1←Y $1) will replace the third element of X with the value of Y. As with assignments, replacements are not performed until after it is determined that the entire match will be successful.

Replacements involving segments splice the corresponding structure into the list being matched, e.g., if X is (A B C D E F) and FOO is (1 2 3), after the pattern ('A $←FOO 'D $) is matched with X, X will be (A 1 2 3 D E F), and FOO will be EQ to CDR of X, i.e., (1 2 3 D E F).

Note that ($ FOO←FIE $) is ambiguous, since it is not clear whether FOO or FIE is the pattern element, i.e., whether ← specifies assignment or replacement. For example, if PATVARDEFAULT is =, this pattern can be interpreted as ($ FOO←=FIE $), meaning search for the value of FIE, and if found set FOO to it, or ($ =FOO←FIE $) meaning search for the value of FOO, and if found, store the value of FIE into the corresponding position. In such cases, the user should disambiguate by not using the PATVARDEFAULT option, i.e., by specifying ' or =.

Note: Replacements are normally done with RPLACA or RPLACD. The user can specify that /RPLACA and /RPLACD should be used, or FRPLACA and FRPLACD, by means of CLISP declarations (see page 16.9).

23.1.7 Reconstruction

The user can specify a value for a pattern match operation other than what is returned by the match by writing FORM_1:*PATTERN*=>FORM_2.[4] For example, X:(FOO←$ 'A --) => (REVERSE FOO) translates as:

```
[PROG ($$2)
   (RETURN
      (COND ((SETQ $$2 (MEMB 'A X))
             (SETQ FOO (LDIFF X $2))
             (REVERSE FOO]
```

Place-markers in the pattern can be referred to from within *FORM*, e.g., the above could also have been written as X:(!#1 'A --)=>(REVERSE #1). If -> is used in place of =>, the expression being

[4] The original CLISP is replaced by an expression of the form (MATCH FORM_1 WITH *PATTERN* => FORM_2). CLISP also recognizes expressions input in this form.

matched is also *physically changed* to the value of *FORM*. For example, `X:(#1 'A !#2) -> (CONS #1 #2)` would remove the second element from `X`, if it were equal to `A`.

In general, *FORM*$_1$:*PATTERN*->*FORM*$_2$ is translated so as to compute *FORM*$_2$ if the match is successful, and then smash its value into the first node of *FORM*$_1$. However, whenever possible, the translation does not actually require *FORM*$_2$ to be computed in its entirety, but instead the pattern match compiler uses *FORM*$_2$ as an indication of what should be done to *FORM*$_1$. For example, `X:(#1 'A !#2) -> (CONS #1 #2)` translates as `(AND (EQ (CADR X) 'A) (RPLACD X (CDDR X)))`.

23.1.8 Examples

`X:(-- 'A --)`

-- matches any arbitrary segment. `'A` matches only an `A`, and the second `--` again matches an arbitrary segment; thus this translates to `(MEMB 'A X)`.

`X:(-- 'A)`

Again, `--` matches an arbitrary segment; however, since there is no `--` after the `'A`, `A` must be the last element of `X`. Thus this translates to: `(EQ (CAR (LAST X)) 'A)`.

`X:('A 'B -- 'C $3 --)`

`CAR` of `X` must be `A`, and `CADR` must be `B`, and there must be at least three elements after the first `C`, so the translation is:

```
(AND (EQ (CAR X) 'A)
     (EQ (CADR X) 'B)
     (CDDDR (MEMB 'C (CDDR X))))
```

`X:(('A 'B) 'C Y←$1 $)`

Since `('A 'B)` does not end in `$` or `--`, `(CDDAR X)` must be `NIL`.

```
(COND
   ((AND (EQ (CAAR X) 'A)
         (EQ (CADAR X) 'B)
         (NULL (CDDAR X))
         (EQ (CADR X) 'C)
         (CDDR X))
    (SETQ Y (CADDR X))
    T))
```

`X:(#1 'A $ 'B 'C #1 $)`

`#1` is implicitly assigned to the first element in the list. The `$` searches for the first `B` following `A`. This `B` must be followed by a `C`, and the `C` by an expression equal to the first element.

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CADR X) 'A)
           (EQ [CADR (SETQ $$2 (MEMB 'B (CDDR X] 'C)
           (CDDR $$2)
           (EQUAL (CADDR $$2) (CAR X]
```

`X:(#1 'A -- 'B 'C #1 $)`

Similar to the pattern above, except that -- specifies a search for *any* B followed by a C followed by the first element, so the translation is:

```
[AND (EQ (CADR X) 'A)
     (SOME (CDDR X)
           (FUNCTION (LAMBDA ($$2 $$1)
              (AND (EQ $$2 'B)
                   (EQ (CADR $$1) 'C)
                   (CDDR $$1)
                   (EQUAL (CADDR $$1) (CAR X]
```

23.2 PRINTING REENTRANT AND CIRCULAR LIST STRUCTURES

23.2.1 CIRCLPRINT

Note: CIRCLPRINT is a LispUsers package contained on the file CIRCLPRINT.DCOM.

HPRINT (page 6.24) is designed primarily for dumping circular or reentrant list structures (as well as other data structures for which READ is not an inverse of PRINT) so that they can be read back in by Interlisp. The CIRCLPRINT package is designed for printing circular or reentrant structures so that the user can look at them and understand them.

A reentrant list structure is one that contains more than one occurrence of the same (EQ) structure. For example, TCONC (page 2.17) makes uses of reentrant list structure so that it does not have to search for the end of the list each time it is called. Thus, if X is a list of 3 elements, (A B C), being constructed by TCONC, the reentrant list structure used by TCONC for this purpose is:

```
-----
|.|.|-----------------|
-----                 |
 |                    |
 V                    V
-----     -----     -----
|A|.|---->|B|.|---->|C|/|
-----     -----     -----
```

This structure would be printed by PRINT as ((A B C) C). Note that PRINT would produce the same output for the non-reentrant structure:

```
-----     -----
|.|.|---->|C|/|
-----     -----
 |
 V
-----     -----     -----
|A|.|---->|B|.|---->|C|/|
-----     -----     -----
```

In other words, PRINT does not indicate the fact that portions of the structure in the first figure are identical. Similarly, if PRINT is applied to a circular list structure (a special type of reentrant structure) it will never terminate.

For example, if PRINT is called on the structure:

```
.    -----
|--->|.|/|
|    -----
|     |
|-----|
```

it will print an endless sequence of left parentheses, and if applied to:

```
.    -----
|--->|A|.|----|
|    -----    |
|             |
|-------------|
```

will print a left parenthesis followed by an endless sequence of A's.

The function CIRCLPRINT described below produces output that will exactly describe the structure of any circular or reentrant list structure. This output may be in either single or double-line formats. Below are a few examples of the expressions that CIRCLPRINT would produce to describe the structures discussed above.

First Figure, single line:

```
((A B *1* C) {1})
```

First Figure, double-line:

```
((A B  C) {1})
     1
```

Third Figure, single-line:

```
(*1* {1})
```

Third Figure, double-line:

```
({1})
1
```

Forth Figure, single-line:

```
(*1* A . {1})
```

Forth Figure, double-line:

```
(A . {1})
1
```

The more complex structure:

```
          -----
|-------->|.|.|--------------------------|
|         -----                          |
|          |                             |
|          V                             V
|         -----     -----     -----     -----
|    |--->|.|.|---->|.|.|---->|A|.|---->|B|.|
|    |    -----     -----     -----     -----
|    |     |         |↑                  |
|    |-----|         ||------------------|
|                    |
|--------------------|
```

is printed as follows:

Single-line:

```
(*2* (*1* {1} *3* {2} A *4* B . {3}) . {4})
```

Double-line:

```
(({1}  {2} A   B . {3}) . {4})
21     3    4
```

In both formats, the reentrant nodes in the list structure are labeled by numbers. (A reentrant node is one that has two or more pointers coming into it.) In the single-line format, the label is printed between asterisks at the beginning of the node (list or tail) that it identifies. In the double-line format, the label is printed below the beginning of the node it identifies. An occurrence of a reentrant node that has already been identified is indicated by printing its label in brackets.

(CIRCLPRINT *LIST* *PRINTFLG* *RLKNT*) [Function]

Prints an expression describing *LIST*. If *PRINTFLG*=NIL, double-line format is used, otherwise single-line format. CIRCLPRINT first calls CIRCLMARK, and then calls either RLPRIN1 (if *PRINTFLG*=T) or RLPRIN2 (if *PRINTFLG*=NIL). Finally, RLRESTORE is called to restore *LIST* to its unmarked state. Returns *LIST*.

(CIRCLMARK *LIST* *RLKNT*\ [Function]

Mar ach reentrant node in *LIST* with a unique number, starting at *RLKNT*+1 (or *RLKNT* is NIL). Value is (new) *RLKNT*.

Marking *LIST* physically alters it. However, the marking is performed undoably. In addition, *LIST* can always be restored by specifically calling RLRESTORE.

(RLPRIN1 *LIST*) [Function]

Prints an expression describing *LIST* in the single-line format. Does not restore *LIST* to its unCIRCLMARKed state. *LIST* must previously have been CIRCLMARKed or an error is generated.

(RLPRIN2 *LIST*) [Function]

Same as RLPRIN1, except that the expression describing *LIST* is printed in the double-line format.

(RLRESTORE *LIST*) [Function]

Physically restores list to its original, unmarked state.

Note that the user can mark and print several structures which together share common substructures, e.g., several property lists, by making several calls to CIRCLMARK, followed by calls to RLPR-IN1 or RLPRIN2, and finally to RLRESTORE.

(CIRCLMAKER *LIST*) [Function]

LIST may contain labels and references following the convention used by CIRCLPRINT for printing reentrant structures in single line format, e.g., (*1* . {1}). CIRCLMAKER performs the necessary RPLACA's and RPLACD's to make *LIST* correspond to the indicated structure. Value is (altered) *LIST*.

(CIRCLMAKER1 *LIST*) [Function]

Does the work for CIRCLMAKER. Uses free variables LABELST and REFLST. LABELST is a list of dotted pairs of labels and corresponding nodes. REFLST is a list of nodes containing references to labels not yet seen. CIRCLMAKER operates by initializing LABELST and REFLST to NIL, and then calling CIRCLMAKER1. It generates an error if REFLST is not NIL when CIRCLMAKER1 returns. The user can call CIRCLMAKER1 directly to "connect up" several structures that share common substructures, e.g., several property lists.

23.2.2 PRINTL

Note: PRINTL is a LispUsers package contained on the file PRINTL.COM.

The PRINTL package uses a different scheme than CIRCLPRINT to present circular structures in an easily readable format. PRINTL uses indentation a la PRETTYPRINT to make it easier for the user to see the nesting of list structure, and prints index numbers for the beginning and ends of expressions so that the user can find what is referred back to easily. Note that PRINTL does not provide an output format which can be read back in to reconstruct the original list structure; it is intended primarily as a debugging aid.

The following example illustrates the use of PRINTL:

```
32←(PRINTL (NCONC (SETQQ X (A B C D)) X))
  1: (A B C D . {1})                              :1
NIL
33←(PRINTL (LIST X (CDR X) (CDDR X) (CDDDR X]
  1: ((A B C D . {2}) {3} {4} {5})                :1
NIL
34←(PRINTL (LIST X (CONS 'P (CDR X)) (CONS 'Q (CDDR X))
(CONS 'R (CDDDR X]
  1: ((A B C D . {2})                             :2
  6: (P . {3})                                    :6
  7: (Q . {4})                                    :7
  8: (R . {5}))                                   :1
NIL
35←USE LIST FOR CONS
  1: ((A B C D . {2})                             :2
  6: (P {3})                                      :6
  8: (Q {4})                                              :8
  10  (R {5}))                                            :1
NIL
```

PRINTL uses the following algorithm: Each list node that is printed (CAR or CDR) is assigned a number. The second and subsequent appearences of this list node are represented simply by printing the number corresponding to the node in {} brackets. Every line on which the representation of a list begins shows the corresponding number of the *first* such list, i.e. this number corresponds to the first open parenthesis on the line. Similarly, to the right of every line on which a list ends is printed the number that corresponds to the *last* close parenthesis on the line. The numbers for those list nodes which do not correspond to the first open parentheses or the last close parentheses on a line can be obtained by simply counting from the last numbered parenthesis. For example, in the line

```
1: ((A B C D . {2}) {3} {4} {5})                          :1
```

2 is (A B C D), 3 is (B C D), 4 is (C D), and 5 is (D).

(PRINTL *ITEM DEPTH LMARG RMARG FILE*) [Function]

Prints an item which is known to be, or suspected of being a circular list structure, in the form described above. *DEPTH* controls the depth of recursion in the CAR direction and defaults to the value of the varible PRINTDEPTH (initially 4). Elements of the structure at this depth are printed as "{--}".

LMARG is the left margin. If NIL, *LMARG* defaults to (POSITION *FILE*). *RMARG* is the position at which the righthand column of numbers will be printed. If NIL, *RMARG* defaults to (LINELENGTH)-5.

Printing is to *FILE*, which is opened if necessary.

PRINTDEPTH [Variable]

The default *DEPTH* argument for PRINTL. Initially 4.

(PRNTL *ARGS*) [Prog. Asst. Command]

Programmers Assistant command that performs (PRINTL . *ARGS*) provided (CAR *ARGS*) is not a number. If it is, or if *ARGS*=NIL, the item to be printed is taken to be the last event on the history list with a non-null value. Thus PRNTL 6 will print the last non-null value with *DEPTH*=6.

23.3 INDEXING AND CROSS REFERENCING FILES

23.3.1 SINGLEFILEINDEX

Note: SINGLEFILEINDEX is a LispUsers package that is contained on the file SINGLEFILEINDEX.DCOM.

SINGLEFILEINDEX is a package for giving the user an alphabetical function index on the front of each lisp file listed by Interlisp. This package is similar to the MULTIFILEINDEX package described below, except that SINGLEFILEINDEX provides a table of contents for functions only, and operates on one file at a time. However, SINGLEFILEINDEX is much simpler and faster than MULTIFILEINDEX and is useful every time a file is made.

The first page gives the filename, time of creation, and the time of the listing. Following that (on possibly more than one page) are *N* columns of function names and index numbers, where the index number indicates the function's linear occurrence within the file. The number of columns is determined by the length of the longest function name, as well as by the number of functions in the file as described below. The file is then printed with the filename and page number at the top of every page, and each function is preceded by its index number right-justified on the page.

When the SINGLEFILEINDEX package is first loaded, it redefines `LISTFILES1` (page 11.9) so that all files listed by `LISTFILES` will be listed by calling `(SINGLEFILEINDEX` *FILE* `NIL NIL)`. Note that the file being indexed does not have to be loaded, or even noticed in the file package sense.

`(SINGLEFILEINDEX` *FILE OUTPUTFILE NEWPAGEFLG*`)` [Function]

FILE is the lisp source file. *OUTPUTFILE* is the destination file. If *OUTPUTFILE* = `NIL`, then the value of `PRINTER` (initially `LPT:`) is used. *NEWPAGEFLG* = `T` means each function will be printed on a new page. The value of `FILELINELENGTH` determines the position of the index numbers, as well as the placement of the columns. The value of `LINESPERPAGE` (initially 58) determines the number of lines per page.

23.3.2 MULTIFILEINDEX

Note: MULTIFILEINDEX is a LispUsers package that is contained on the file `MULTIFILEINDEX.DCOM`.

Many systems built in Interlisp consist of a number of symbolic source files. Finding one's way around in the listings can often be very tedious, even for the implementor of the system, if you don't know the system and the structure of the files intimately. The MULTIFILEINDEX package is an attempt to help users deal with this problem by creating a listing of an entire system or set of files, including an alphabetized table of contents containing entries for each function on any of the files. Information (but not unique index numbers) is included for other entities in the files such as records, blocks, and properties. The function `MULTIFILEINDEX` implements this mechanism.

`(MULTIFILEINDEX` *SOURCEFILES DESTINATIONFILE NEWPAGEFLG*`)` [Function]

SOURCEFILES is a list of file names (if atomic, `(LIST` *SOURCEFILES*`)` is used). If it is `NIL`, `MULTIFILEINDEX` returns immediately. If it is `T`, the value of `FILELST` is used (page 11.13). *DESTINATIONFILE* is the output file. If *DESTINATIONFILE* is `NIL`, the value of `PRINTER` is used (below). If *NEWPAGEFLG* = `T`, each function in the listing will be placed on a page by itself.

In the default case, `MULTIFILEINDEX` does the following:

(1) Outputs an alphabetized table of contents (index) indicating the name of an object (function, record, block, variable, and so on), the file that it belongs to, and its type (property, variable (set or saved), record, block, and so forth). If the object is the name of a function, then the information includes a unique index in the listing for the function, its type (`EXPR`, `FEXPR*`, etc.), and its argument list. Note that it handles functions/files that use DECL (page 23.18). Otherwise, the index represents the index of the function immediately preceeding the definition of the entity.

(2) Outputs a listing of the files with each function being preceeded by its index number right-justified

on the line. Header information is placed at the top of each page, and the pages are numbered.

(3) Undoably removes the names of the files indexed from NOTLISTEDFILES (page 11.9).

MULTIFILEINDEX is effected by the following variables:

MULTIFILEINDEXMAPFLG [Variable]

If T, indicates that you want the file index output. Initially T.

MULTIFILEINDEXFILESFLG [Variable]

If T, indicates that you want the file listings output to *DESTINATIONFILE*. Initially T.

PRINTER [Variable]

If the *MAPFILE* argument to MULTIFILEINDEX is NIL, it defaults to the value of PRINTER. Initially {LPT} in Interlisp-D, LPT: in Interlisp-10.

LINESPERPAGE [Variable]

The value of LINESPERPAGE determines the number of lines per page. Initially 65 in Interlisp-D, 58 in Interlisp-10.

FONTCHANGEFLG [Variable]

If NIL, page headings and the index numbers that preceed the definition of each function are printed bold; that is, overprinted; otherwise, they are printed using the BOLDFONT (PRETTYCOMFONT if BOLDFONT doesn't exist) in the current FONTPROFILE (see page 6.55).

FILELINELENGTH [Variable]

The value of FILELINELENGTH determines the width of the page.

The following four parameters affect how the columns are placed:

MULTIFILEINDEXCOLS [Variable]
MULTIFILEINDEXNAMECOL [Variable]
MULTIFILEINDEXFILECOL [Variable]
MULTIFILEINDEXTYPECOL [Variable]

The value of MULTIFILEINDEXCOLS indicates how the other three are to be interpreted. If MULTIFILEINDEXCOLS is the atom FLOATCOLS (its initial value), then an attempt is made to fit the columns onto the page in a way that maximizes the amount of space for the type information (the amount of space allocated for the type field must be at least 45% of FILELINELENGTH in this case). If MULTIFILEINDEXCOLS is either T or FIXCOLS, then the value of the other variables are treated as absolute column positions on the page. If MULTIFILEINDEXCOLS is either NIL or FIXFLOATCOLS, the columns will be floated, but will not be any smaller than the column positions defined by the other variables.

The initial values of these four variables are FLOATCOLS, 0, 26 and 41, respectively.

MULTIFILEINDEX has an interface to Masterscope. If the value of either of the next two variables is T, then MULTIFILEINDEX assumes that the sourcefiles have already been analyzed by Masterscope, and calls UPDATECHANGED.

MULTIFILEINDEXFNSMSFLG [Variable]

If T, indicates that you want the Masterscope information about each function output. This includes who calls each function, who this function calls, and what variables are set or referred to either locally or freely. Initially NIL.

MULTIFILEINDEXVARSMSFLG [Variable]

If T, indicates that all variables used in the files should have some information output about them at the end of the listing. The list of variables to look at is obtained by effectively asking Masterscope the question: "WHO IS USED BY ANY AND WHO IS SET BY ANY". The listing will include information about who binds, uses freely or locally, or smashes freely or locally each variable. The variable map is case-independently sorted by the name of the variable. Initially NIL.

In order to make the index, or map, of the files, the filecoms for all the files being listed need be loaded (see page 11.21); MULTIFILEINDEX does a GETDEF on each file (file names are obtained using FINDFILE) to obtain its filecoms. As other indirections are noted, they also are obtained using GETDEF. For example, if you have a file TEST, and its filecoms is ((FNS * TESTFNS)), just doing a GETDEF on TESTCOMS will not suffice; as the expression (FNS * TESTFNS) is parsed, a GETDEF is also done to obtain the value of TESTFNS.

MULTIFILEINDEXLOADVARSFLG [Variable]

If T, then a LOADVARS of all the VARS on a particular file is performed before the filecoms is loaded with GETDEF. Initially NIL.

MULTIFILEINDEXGETDEFFLG [Variable]

If T, MULTIFILEINDEX will inform the user when it does GETDEFs. Initially NIL.

23.4 DATABASEFNS

Note: Databasefns is a LispUsers package that is contained on the file DATABASEFNS.DCOM.

Databasefns is a very small package whose purpose is to make the construction and maintenance of MASTERSCOPE databases an essentially automatic process. It modifies MAKEFILE, LOAD, and LOADFROM to behave in the following way:

A database will be maintained automatically for any file (containing functions) whose file name has the property DATABASE with value YES. Whenever such a file is dumped via MAKEFILE, MASTERSCOPE will analyse any new or changed functions on the file, and a database for all of the functions on the file will be written on a separate file whose name is of the form *FILE*.DATABASE. Whenever a file which has a DATABASE property with value YES is loaded via LOAD or LOADFROM, then the corresponding .DATABASE file, if any, is also loaded. The database will not be dumped or loaded if the value of the DATABASE property for the file is NO. The DATABASE property is considered to be NO if the file is loaded with *LDFLG*=SYSLOAD.

If the DATABASE property is not YES or NO, then for MAKEFILE, LOAD, and LOADFROM will ask the user whether he wants automatic database maintenance. The user's answer will be stored on the DATABASE property so that he will not be asked again. Thus when a file is dumped for the first time, the user will be asked "Do you want a Masterscope Database for this file?". Similarly, if the user loads a file which has an associated database, the user will be asked "load database for *FILE*?".

The above interactions may be controlled via the global variables SAVEDBFLG and LOADDBFLG. When a file which has neither a YES or NO database property is being dumped, MAKEFILE will assume (and store) a YES value if the value of SAVEDBLFG is YES, and a NO value if SAVEDBFLG is NO. The user will be queried only if SAVEDBFLG is ASK (its initial value). Similarly, if LOADDBFLG is YES, LOAD and LOADFROM will automatically load an existing .DATABASE file for a file which does not have a YES or NO value for its DATABASE property. The database will not be loaded if LOADDBFLG is NO, and the user will be interrogated as described above if LOADDBFLG is ASK (its initial value).

The user can dump and restore databases explicitly via the following functions:

(DUMPDB *FILE*) [Function]

Dumps a database for *FILE* then sets the DATABASE property to YES, so that database maintenance for *FILE* will subsequently be automatic.

(LOADDB *FILE*) [Function]

Loads the file *FILE*.DATABASE if one exists. After the database is loaded, the DATABASE property for *FILE* is set to YES, so that maintenance will thereafter be automatic.

Database files include the date and full filename of the file to which they correspond. LOADDB will print out a warning message if it loads a database that does not correspond to the in-core version of the file.

Note that LOADDB is the only approved way of loading a database: Attempting to load a database file will cause an error.

23.5 LAMBDATRAN

Note: Lambdatran is a LispUsers package that is contained on the file LAMBDATRAN.DCOM.

The purpose of this package is to facilitate defining new LAMBDA words in such a way that a variety of other system packages will respond to them appropriately. A LAMBDA word is a word that can appear as CAR of a function definition, like LAMBDA and NLAMBDA. New LAMBDA words are useful because they enable the user to define his own conventions about such things as the interpretation of arguments, and to build in certain defaults about how values are returned. For example, the DECL package (page 23.18) defines DLAMBDA as a new LAMBDA word with unconventional arguments such as the following:

```
(DLAMBDA ((A FLOATP) (B FIXP) (RETURNS SMALLP)) (FOO A B))
```

In order for such an expression to be executable and compilable, a mechanism must be provided for translating this expression to an ordinary LAMBDA or NLAMBDA, with the special behavior associated with the arguments built into the function body. The lambdatran package accomplishes this via an appropriate entry on DWIMUSERFORMS (see page 15.10) that computes the translation.

Besides executing and compiling, Interlisp applies a number of other operations to function definitions (e.g. breaking, advising), many of which depend on the system being able to determine certain properties of the function, such as the names of its arguments, their number, and the type of the function (EXPR, FEXPR, etc.). The lambdatran package also provides new definitions for the functions FNTYP, ARGLST, NARGS, and ARGTYPE which can be told how to compute properties for the user's LAMBDA-words.

A new LAMBDA-word is defined in the following way:

1. Add the LAMBDA-word itself (e.g. the atom DLAMBDA) to the list LAMBDASPLST. This suppresses attempts to correct the spelling of the LAMBDA-word.

2. Add an entry for the LAMBDA-word to the association-list LAMBDATRANFNS, which is a list of elements of the form: (*LAMBDA-WORD TRANFN FNTYP ARGLIST*), where

LAMBDA-WORD is the name of the LAMBDA-word (e.g. DLAMBDA).

TRANFN is a function of one argument that will be called whenever a real definition is needed for the LAMBDA-word definition. Its argument is the LAMBDA-word definition, and its value should be a conventional LAMBDA or NLAMBDA expression which will become the translation of the LAMBDA-word form. The free variable FAULTFN is bound to the name of the function in which the LAMBDA-word form appeared (or TYPE-IN if the form was typed in).

FNTYP determines the function-type of a definition beginning with *LAMBDA-WORD*. It is consulted if the definition does not already have a translation from which the function type may be deduced. If *FNTYP* is one of the atoms EXPR, FEXPR, EXPR*, FEXPR*, then all definitions beginning with LAMBDA-word are assumed to have that type. Otherwise, *FNTYP* is a function of one argument that will be applied to the LAMBDA-word definition. Its value should be one of the above four function types.

ARGLIST determines the argument list of the definition if it has not already been translated (if it has, the *ARGLIST* is simply the *ARGLIST* of the translation). It is also a function of one argument, the LAMBDA-word definition, and its value should be the list of arguments for the function (e.g. (A B) in the DLAMBDA example above). If the LAMBDA-word definition is ill-formed and the argument list cannot be computed, the function should return T. If an *ARGLIST* entry is not provided in the LAMBDATRANFNS element, then the argument list defaults to the second element of the definition.

As an example, the LAMBDATRANFNS entry for DLAMBDA is (DLAMBDA DECL EXPR DLAMARGLIST), where DECL and DLAMARGLIST are functions of one argument.

Note: if the LAMBDA-word definition has an argument list with argument names appearing either as literal atoms or as the first element of a list, the user should also put the property INFO with value BINDS on the property list of the LAMBDA-word in order to inform DWIMIFY (page 16.14) to take notice of the names of the arguments when DWIMIFYing.

23.6 PERMSTATUS

Note: Permstatus is a LispUsers package that is contained on the file PERMSTATUS.COM.

The function PERMSTATUS defined in this package can be used in conjunction with WHENCLOSE (page 6.11) to make a file "permanently" open in the sense that as much of its status as possible will be restored when a SYSOUT is resumed. This includes its access mode, file-pointer position, bytesize, and any pages mapped in by the page mapping facility (page 14.17). The desired effect is achieved by saying

(WHENCLOSE *FILENAME* 'STATUS 'PERMSTATUS) after the file has been opened.

Note that the permanency of files is not guaranteed in that files may be deleted or renamed, or their contents changed, despite their permanent attribute in some SYSOUT. When restarting a SYSOUT, a warning message will be printed if the file cannot be found or restored. However, PERMSTATUS will not be able to detect that the contents of a file have been modified since the SYSOUT was created. Note also that "permanent" files will still be closed by CLOSEF, and will not be immune to CLOSEALL or to closing on end-of-file errors unless the appropriate WHENCLOSE attributes for CLOSEALL and EOF are also established.

23.7 THE DECL PACKAGE

Note: Decl is a LispUsers package that is contained on the file DECL.DCOM. *The Decl package requires the* LAMBDATRAN *package (section 23.5), so* LAMBDATRAN.DCOM *will automatically be loaded with Decl if it is not already present.*

The Decl package extends Interlisp to allow the user to declare the types of variables and expressions appearing in functions. It provides a convenient way of constraining the behavior of programs when the generality and flexibility of ordinary Interlisp is either unnecessary, confusing, or inefficient.

The Decl package provides a simple language for declarations, and augments the interpreter and the compiler to guarantee that these declarations are always satisfied. The declarations make programs more readable by indicating the type, and therefore something about the intended usage, of variables and expressions in the code. They facilitate debugging by localizing errors that manifest themselves as type incompatibilities. Finally, the declaration information is available for other purposes: compiler macros can consult the declarations to produce more efficient code; coercions for arguments at user interfaces can be automatically generated; and the declarations will be noticed by the Masterscope function analyzer.

The declarations interpreted by the Decl package are in terms of a set of declaration types called *decltypes*, each of which specifies a set of acceptable values and also (optionally) other type specific behavior. The Decl package provides a set of facilities for defining decltypes and their relations to each other, including type valued expressions and a comprehensive treatment of union types.

The following description of the Decl package is divided into three parts. First, the syntactic extensions which permit the concise attachment of declarations to program elements are discussed. Second, the mechanisms by which new decltypes can be defined and manipulated are covered. Finally, some additional capabilities based on the availability of declarations are outlined.

23.7.1 Using Declarations in Programs

Declarations may be attached to the values of arbitrary expressions and to LAMBDA and PROG variables throughout (or for part of) their lexical scope. The declarations are attached using constructs that resemble the ordinary Interlisp LAMBDA, PROG, and PROGN, but which also permit the expression of declarations. The following examples illustrate the use of declarations in programs.

Consider the following definition for the factorial function (FACT *N*):

```
[LAMBDA (N)
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

Obviously, this function presupposes that N is a number, and the run-time checks in ITIMES and SUB1 will cause an error if this is not so. For instance, (FACT T) will cause an error and print the message NON-NUMERIC ARG T. By defining FACT as a DLAMBDA, the Decl package analog of LAMBDA, this presupposition can be stated directly in the code:

```
[DLAMBDA ((N NUMBERP))
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

With this definition, (FACT T) will *not* result in a NON-NUMERIC ARG T error when the body of the code is executed. Instead, the NUMBERP declaration will be checked when the function is first entered, and a *declaration fault* will occur. Thus, the message that the user will see will not dwell on the offending value T, but instead give a symbolic indication of what variable and declaration were violated, as follows:

```
DECLARATION NOT SATISFIED
((N NUMBERP) BROKEN)
:
```

The user is left in a break from which the values of variables, e.g. N, can be examined to determine what the problem is.

The function FACT also makes other presuppositions concerning its argument, N. For example, FACT will go into an infinite recursive loop if N is a number less than zero. Although the user could program an explicit check for this unexpected situation, such coding is tedious and tends to obscure the underlying algorithm. Instead, the requirement that N not be negative can be succinctly stated by declaring it to be a subtype of NUMBERP which is restricted to non-negative numbers. This can be done by adding a SATISFIES clause to N's type specification:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N])
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

The predicate in the SATISFIES clause will be evaluated after N is bound and found to satisfy NUMBERP, but before the function body is executed. In the event of a declaration fault, the SATISFIES condition will be included in the error message. For example, (FACT -1) would result in:

```
DECLARATION NOT SATISFIED
((N NUMBERP (SATISFIES (NOT (MINUSP N))) BROKEN)
:
```

The DLAMBDA construct also permits the type of the value that is returned by the function to be declared by means of the pseudo-variable RETURNS. For example, the following definition specifies that FACT is to return a positive integer:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
          [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

After the function body is evaluated, its value is bound to the variable VALUE and the RETURNS declaration is checked. A declaration fault will occur if the value is not satisfactory. This prevents a bad value from propagating to the caller of FACT, perhaps causing an error far away from the source of the difficulty.

Declaring a variable causes its value to be checked not only when it is first bound, but also whenever that variable is reset by SETQ within the DLAMBDA. In other words, the type checking machinery will not allow a declared variable to take on an improper value. An iterative version of the factorial function illustrates this feature in the context of a DPROG, the Decl package analog of PROG:

```
(DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
          [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
   [DPROG ([TEMP 1 FIXP (SATISFIES (IGREATERP TEMP 0]
           [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
      LP  (COND ((EQ N 0) (RETURN TEMP)))
          (SETQ TEMP (ITIMES N TEMP))
          (SETQ N (SUB1 N))
          (GO LP]
```

DPROG declarations are much like DLAMBDA declarations, except that they also allow an initial value for the variable to be specified. In the above example, TEMP is declared to be a positive integer throughout the computation and N is declared to be non-negative. Thus, a bug which caused an incorrect value to be assigned by one of the SETQ expressions would cause a declaration failure. Note that the RETURNS declaration for a DPROG is also useful in detecting the common bug of omitting an explicit RETURN.

23.7.2 DLAMBDAs

The Decl package version of a LAMBDA expression is an expression beginning with the atom DLAMBDA. Such an expression is a function object that may be used in any context where a LAMBDA expression may be used. It resembles a LAMBDA expression except that it permits declaration expressions in its argument list, as illustrated in the examples given earlier. Each element of the argument list of a DLAMBDA may be a literal atom (as in a conventional LAMBDA) or a list of the form (*NAME TYPE . EXTRAS*).[5]

NAME fulfills the standard function of a parameter, i.e. providing a name to which the value of the corresponding argument will be bound.

TYPE is either a Decl package type name or type expression. When the DLAMBDA is entered, its arguments will be evaluated and bound to the corresponding argument names, and then, after *all* the argument names

[5] Strictly, this would require a declaration with a SATISFIES clause to take the form (N (NUMBERP (SATISFIES --)) --) (page 23.27). However, due to the frequency with which this construction is used, it may be written without the inner set of parentheses, e.g. (N NUMBERP (SATISFIES --) --).

have been bound, the declarations will be checked. The type checking is delayed so that SATISFIES predicates can include references to other variables bound by the same DLAMBDA. For example, one might wish to define a function whose two arguments are not only both required to be of some given type, but are also required to satisfy some relationship (e.g., that one is less than the other).

EXTRAS allows some additional properties to be attached to a variable. One such property is the accessibility of *NAME* outside the current lexical scope. Accessibility specifications include the atoms LOCAL or SPECIAL, which indicate that this variable is to be compiled so that it is either a LOCALVAR or a SPECVAR, respectively. This is illustrated by the following example:

```
[DLAMBDA ((A LISTP SPECIAL)
          (B FIXP LOCAL))
         ...]
```

A more informative equivalent to the SPECIAL key word is the USEDIN form, the tail of which can be a list of the other functions which are expected to have access to the variable:[6]

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE))
          (B FIXP LOCAL))
         ...]
```

EXTRAS may also include a comment in standard format, so that descriptive information may be given where a variable is bound:

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE)   (* This is an important variable))
          (B FIXP LOCAL))
         ...]
```

As mentioned earlier, the value returned by a DLAMBDA can also be declared, by means of the pseudo-variable RETURNS. The RETURNS declaration is just like other DLAMBDA declarations, except (1) in any SATISFIES predicate, the value of the function is referred to by the distinguished name VALUE; and (2) it makes no sense to declare the return value to be LOCAL or SPECIAL.

23.7.3 DPROG

Just as DLAMBDA resembles LAMBDA, DPROG is analogous to PROG. As for an ordinary PROG, a variable binding may be specified as an atom or a list including an initial value form. However, a DPROG binding also allows *TYPE* and *EXTRAS* information to appear following the initial value form. The format for these augmented variable bindings is (*NAME INITIALVALUE TYPE* . *EXTRAS*). The only difference between a DPROG binding and a DLAMBDA binding is that the second position is interpreted as the initial value for the variable. Note that if the user wishes to supply a type declaration for a variable, an initial value *must* be specified. The same rules apply for the interpretation of the type information for DPROGs as for DLAMBDAs, and the same set of optional *EXTRAS* can be used. DPROGs may also declare the type of the value they return, by specifying the pseudo-variable RETURNS.

[6] USEDIN is mainly for documentation purposes, since there is no way for such a restriction to be enforced.

Just as for a DLAMBDA, type tests in a DPROG are not asserted until *after* all the variables have been bound, thus permitting predicates to refer to other variables being bound by this DPROG. If NIL appears as the initial value for a binding (i.e. the atom NIL actually appears in the code, not simply an expression which evaluates to NIL) the initial type test will be suppressed, but subsequent type tests, e.g. following a SETQ, will still be performed.

A common construct in Lisp is to bind and initialize a PROG variable to the value of a complicated expression in order to avoid recomputing it, and then to use this value in initializing other PROG variables, e.g.

```
[PROG ((A EXPRESSION))
      (RETURN (PROG ((B (··· A ···))
                     (C (··· A ···)))
                    ···]
```

The ugliness of such constructions in conventional Lisp often tempts the programmer to loosen the scoping relationships of the variables by binding them all at a single level and using SETQ's in the body of the PROG to establish the initial values for variables that depend on the initial values of other variables, e.g.

```
[PROG ((A EXPRESSION) B C)
      (SETQ B (··· A ···))
      (SETQ C (··· A ···))
      ···]
```

In the Decl package environment, this procedure undermines the protection offered by the type mechanism by encouraging the use of uninitialized variables. Therefore, the DPROG offers a syntactic form to encourage more virtuous initialization of its variables. A DPROG variable list may be segmented by occurrences of the special atom THEN, which causes the binding of its variables in stages, so that the bindings made in earlier stages can be used in later ones, e.g.

```
[DPROG ((A (LENGTH FOO) FIXP LOCAL)
        THEN (B (SQRT A) FLOATP)
        THEN (C (CONS A B) LISTP))
       ···]
```

Each stage is carried out as a conventional set of DPROG bindings (i.e., simultaneously, followed by the appropriate type testing). This layering of the bindings permits one to gradually descend into a inner scope, binding the local names in a very structured and clean fashion, with initial values type-checked as soon as possible.

23.7.4 Declarations in Iterative Statements

The CLISP iterative statement (page 16.1) provides a very useful facility for specifying a variety of PROGs that follow certain widely used formats. The Decl package allows declarations to be made for the scope of an iterative statement via the DECLARE CLISP i.s.opr. DECLARE can appear as an operator anywhere in an iterative statement, followed by a list of declarations, for example:

```
(for J from 1 to 10 declare (J FIXP) do ···
```

Note that DECLARE declarations do not *create* bindings, but merely provide declarations for existing bindings. For this reason, an initial value cannot be specified and the form of the declaration is the same as that of DLAMBDAs, namely (*NAME TYPE* . *EXTRAS*).

Note that variables bound *outside* of the scope of the iterative statement, i.e. a variable used freely in the i.s, can also be declared using this construction. Such a declaration will only be in effect for the scope of the iterative statement.

23.7.5 Declaring a Variable for a Restricted Lexical Scope

The Decl package also permits declaring the type of a variable over some restricted portion of its existence. For example, suppose the variable X is either a fixed or floating number, and a program branches to treat the two cases separately. On one path X is known to be fixed, whereas on the other it is known to be floating. The Decl package DPROGN construct can be used in such cases to state the type of the variable along each path. DPROGN is exactly like PROGN, except that the second element of the form is interpreted as a list of DLAMBDA format declarations. These declarations are added to any existing declarations in the containing scope, and the composite declaration (created using the ALLOF type expression, page 23.26) is considered to hold throughout the lexical scope created by the DPROGN. Thus, our example becomes:

```
(if (FIXP X)
  then (DPROGN ((X FIXP)) ···)
  else (DPROGN ((X FLOATP)) ···))
```

Like DPROG and DLAMBDA, the value of a DPROGN may also be declared, using the pseudo-variable RETURNS.

DPROGN may be used not only to restrict the declarations of local variables, but also to declare variables which are being used freely. For example, if the variable A is used freely inside a function but is known to be FIXP, this fact could be noted by enclosing the body of the function in (DPROGN ((A FIXP FREE)) *BODY*). Instead of FREE, the more specific construction (BOUNDIN $FUNCTION_1$ $FUNCTION_2$ ···) can be used. This not only states that the variable is used freely but also gives the names of the functions which might have provided this binding.[7]

Since the DPROGN form introduces another level of parenthesization, which results in the enclosed forms being prettyprinted indented, the Decl package also permits such declarations to be attached to their enclosing DLAMBDA or DPROG scopes by placing a DECL expression, e.g. (DECL (A FIXP (BOUNDIN FUM)), before the first executable form in that scope. Like DPROGN's, DECL declarations use DLAMBDA format.

23.7.6 Declaring the Values of Expressions

The Decl package allows the value of an arbitrary form to be declared with the Decl construct THE. A THE expression is of the form (THE *TYPE* . *FORMS*), e.g. (THE FIXP (FOO X)). *FORMS* are evaluated in order, and the value of the *last* one is checked to see if it satisfies *TYPE*, a type name or type expression. If so, its value is returned, otherwise a declaration fault occurs.

[7] Like USEDIN declarations, FREE and BOUNDIN declarations cannot be checked, and are provided for documentation purposes only.

23.7.7 Assertions

The Decl package also allows for checking that an arbitrary predicate holds at a particular point in a program's execution, e.g. a condition that must hold at function entry but not throughout its execution. Such predicates can be checked using an expression of the form (ASSERT *FORM$_1$* *FORM$_2$* ...), in which each *FORM$_i$* is either a list (which will be evaluated) or a variable (whose declaration will be checked). Unless all elements of the ASSERT form are satisfied, a declaration fault will take place.

ASSERTing a variable provides a convenient way of verifying that the value of the variable has not been improperly changed by a lower function. Although a similar effect could be achieved for predicates by explicit checks of the form (OR *PREDICATE* (SHOULDNT)), ASSERT also provides the ability both to check that a variable's declaration is currently satisfied and to remove its checks at compile time without source code modification (see page 23.25).

23.7.8 Using Type Expressions as Predicates

The Decl package extends the Record package TYPE? construct so that it accepts decltypes, as well as record names, e.g. (TYPE? (FIXP (SATISFIES (ILESSP VALUE 0))) *EXPR*). Thus, a TYPE? expression is exactly the same as a THE expression except that, rather than causing a declaration fault, TYPE? is a predicate which determines whether or not the value satisfies the given type.

23.7.9 Enforcement

The Decl package is a "soft" typing system - that is, the data objects themselves are not inherently typed. Consequently, declarations can only be enforced within the lexical scope in which the declaration takes place, and then only in certain contexts. In general, changes to a variable's value such as those resulting from side effects to embedded structure (e.g., RPLACA, SETN, etc.) or free variable references from outside the scope of the declaration cannot be, and therefore are not, enforced.

Declarations *are* enforced i.e. checked, in three different situations: when a declared variable is bound to some value or rebound with SETQ or SETQQ, when a declared expression is evaluated, and when an ASSERT expression is evaluated. In a binding context, the type check takes place *after* the binding, including any user-defined behavior specified by the type's binding function. Any failure of the declarations causes a break to occur and an informative message to be printed. In that break, the name to which the declaration is attached (or *VALUE* if no name is available) will be bound to the offending value. Thus, in the (FACT T) example above, N would be bound to T. The problem can be repaired either by returning an acceptable value from the break via the RETURN command, or by assigning an acceptable value to the offending name and returning from the break via an OK or GO command. The unsatisfied declaration will be reasserted when the computation is continued, so an unacceptable value will be detected.[8]

The automatic enforcement of type declarations is a very flexible and powerful aid to program development. It does, however, exact a considerable run-time cost because of all the checking involved. Factors of two to ten in running speed are not uncommon, especially where low level, frequently used functions employ type declarations. As a result, it is usually desirable to remove the declaration enforcement code when

[8] With this exception, assignments to variables from within the break are not considered to be in the scope of the declarations that were in effect when the break took place, and so are not checked.

the system is believed to be bug-free and performance becomes more central. This can be done with the variable COMPILEIGNOREDECL:

COMPILEIGNOREDECL [Variable]

Setting the value of the variable COMPILEIGNOREDECL to T (initially NIL) instructs the compiler not to insert declaration enforcement tests in the compiled code. More selective removal can be achieved by setting COMPILEIGNOREDECL to a list of function names. Any function whose name is found on this list is compiled without declaration enforcement.

(IGNOREDECL . *VAL*) [File Package Command]

Declaration enforcement may be suppressed selectively by file using the IGNOREDECL file package command. If this appears in a file's file commands, it redefines the value of COMPILEIGNOREDECL to *VAL* for the compilation of this file only.

23.7.10 Decltypes

A Decl package type, or decltype, specifies a subset of data values to which values of this type are restricted. For example, a "positive number" type might be defined to include only those values that are numbers and greater than zero. A type may also specify how certain operations, such as assignment or binding (see page 23.28), are to be performed on variables declared to be of this type.

The inclusion relations among the sets of values which satisfy the different types define a natural partial ordering on types, bound by the universal type ANY (which all values satisfy) and the empty type NONE (which no value satisfies). Each type has one or more *supertypes* (each type has at least ANY as a supertype) and one or more *subtypes* (each type has at least NONE as a subtype). This structure is important to the user of Decl as it provides the framework in which new types are defined. Typically, much of the definition of a new type is defaulted, rather than specified explicitly. The definition will be completed by inheriting atttributes which are shared by all its immediate supertypes.

An initial set of decltypes which defines the Interlisp built-in datatypes and a few other commonly used types is provided. Thereafter, new decltypes are created in terms of existing ones using the type expressions described below. For conciseness, such new types can be associated with literal atoms using the function DECLTYPE (page 23.28).

23.7.11 Predefined Types

Some commonly used types, such as the Interlisp built-in data types, are already defined when the Decl package is loaded. These types, indented to show subtype-supertype relations, are:

```
ANY
  ATOM            LST[9]    ARRAYP      STRINGP    FUNCTION    STACKP
    LITATOM         ALIST[10]  HARRAYP
      NIL           LISTP       READTABLEP
    NUMBERP
      FIXP
        LARGEP
        SMALLP
      FLOATP
```

NONE

Note that the definition of LST causes NIL to have multiple supertypes, i.e. LITATOM and LST, reflecting the duality of NIL as an atom and a (degenerate) list.

In addition, declarations made using the Record package (page 3.1) also define types which are attached as subtypes to an appropriate existing type (e.g., a TYPERECORD declaration defines a subtype of LISTP, a DATATYPE declaration a subtype of ANY, etc.) and may be used directly in declaration contexts.

23.7.12 Type Expressions

Type expressions provide convenient ways for defining new types in terms of modifications to, or compositions of one or more, existing types.

(MEMQ *VALUE*$_1$ ··· *VALUE*$_N$) [Decl Type Expression]

Specifies a type whose values can be any one of the fixed set of elements {*VALUE*$_1$ ··· *VALUE*$_N$}. For example, the status of a device might be represented by a datum restricted to the values BUSY and FREE. Such a "device status" type could be defined via (MEMQ BUSY FREE). The new type will be a subtype of the narrowest type which all of the alternatives satisfy (e.g., the "device status" type would be a subtype of LITATOM). The membership test uses EQ if this supertype is LITATOM; EQUAL otherwise. Thus, lists, floating point numbers, etc., can be included in the set of alternatives.

(ONEOF *TYPE*$_1$ ··· *TYPE*$_N$) [Decl Type Expression]

Specifies a type which is the union of two or more other types. For example, the notion of a possibly degenerate list is something that is either LISTP or NIL. Such a type can be (and the built-in type LST in fact is) defined simply as (ONEOF NIL LISTP). A union data type becomes a supertype of all of the alternative types specified in the ONEOF expression, and a subtype of their lowest common supertype. The type properties of a union type are taken from its alternative types if they all agree, otherwise from the supertype.

(ALLOF *TYPE*$_1$ ··· *TYPE*$_N$) [Decl Type Expression]

Specifies a type which is the intersection of two or more other types. For example, a variable may be required to satisfy both FIXP and also some type which is defined as (NUMBERP (SATISFIES *PREDICATE*)). The latter type will admit numbers that are not FIXP, i.e. floating point numbers; the former does not include *PREDICATE*. Both restrictions can be obtained by using the type (ALLOF (NUMBERP (SATISFIES *PREDICATE*)) FIXP).[11]

[9] LST is defined as either LISTP or NIL, i.e. a list or NIL. The name LST is used, because the name LIST is treated specially by clisp.

[10] ALIST is defined as either NIL, or a list of elements each of which is of type LISTP.

[11] When a value is tested, the component type tests are applied from left to right.

(*AGGREGATE* OF *ELEMENT*) [Decl Type Expression]

Specifies a type which is an aggregate of values of some other type (e.g., list of numbers, array of strings, etc.). *AGGREGATE* must be a type which provides an EVERYFN property (page 23.28). The EVERYFN is used to apply an arbitrary function to each of the elements of a datum of the aggregate type, and check whether the result is non-NIL for each element. *ELEMENT* may be any type expression. For example, the type "list of either strings or atoms" can be defined as (LISTP OF (ONEOF STRINGP ATOM)). The type test for the new type will consist of applying the type test for *ELEMENT* to each element of the aggregate type using the EVERYFN property. The new type will be a subtype of its aggregate type.[12]

(*TYPE* (SATISFIES *FORM*$_1$ ··· *FORM*$_N$)) [Decl Type Expression]

Specifies a type whose values are a subset of the values of an existing type. The type test for the new type will first check that the base type is satisfied, i.e. that the object is a member of *TYPE*, and then evaluate *FORM*$_1$ ··· *FORM*$_N$. If each form returns a non-NIL value, the type is satisfied.

The value that is being tested may be referred to in *FORM*$_1$ ··· *FORM*$_N$ by either (a) the variable name if the type expression appears in a binding context such as DLAMBDA or DPROG (b) the distinguished atom ELT for a SATISFIES clause on the elements of an aggregate type, or (c) the distinguished atom VALUE, when the type expression is used in a context where no name is available (e.g., a RETURNS declaration). For example, one might declare the program variable A to be a negative integer via (FIXP (SATISFIES (MINUSP A))), or declare the value of a DLAMBDA to be of type ((ONEOF FIXP FLOATP) (SATISFIES (GREATERP VALUE 25))). Note that more than one SATISFIES clauses may appear in a single type expression attached to different alternatives in a ONEOF type expression, or attached to both the elements and the overall structure of an aggregate. For example,

```
[LISTP OF [FIXP (SATISFIES (ILEQ ELT (CAR VALUE]
          (SATISFIES (ILESSP (LENGTH VALUE) 7]
```

specifies a list of less than 7 integers each of which is no greater than the first element of the list.

(SHARED *TYPE*) [Decl Type Expression]

Specifies a subtype of *TYPE* with default binding behavior, i.e. the binding function (see page 23.28), if any, will be suppressed.[13] For example, if the type FLOATP were redefined so that DLAMBDA and DPROG bindings of variables that were declared to be FLOATP copied their initial values (e.g., to allow SETNs to be free of side effects), then variables declared (SHARED FLOATP) would be initialized in the normal fashion, without copying their initial values.

[12] The built-in aggregate types are ARRAYP, LISTP, LST, and STRINGP (and their subtypes).

[13] As no predefined type has a binding function, this is of no concern until the user defines or redefines a type to have a binding function.

23.7.13 Named Types

Although type expressions can be used in any declaration context, it is often desirable to save the definition of a new type if it is to be used frequently, or if a more complex specification of its behavior is to be given than is convenient in an expression. The ability to define a named type is provided by the function DECLTYPE.

(DECLTYPE *TYPENAME TYPE PROP$_1$ VAL$_1$* ··· *PROP$_N$ VAL$_N$*) [NLambda NoSpread Function]

Nlambda, nospread function. *TYPENAME* is a literal atom, *TYPE* is either the name of an existing type or a type expression, and *PROP$_1$*, *VAL$_1$*, ···, *PROP$_N$*, *VAL$_N$* is a specification (in property list format) of other attributes of the type. DECLTYPE derives a type from *TYPE*, associates it with *TYPENAME*, and then defines any properties specified with the values given.

The following properties are interpreted by the Decl package.[14] Each of these properties can have as its value either a function name or a LAMBDA expression.

TESTFN — will be used by the Decl package to test whether a given value satisfies this type. The type is considered satisfied if *FN* applied to the item is non-NIL. For example, one might define the type INTEGER with TESTFN FIXP.[15]

EVERYFN — specifies a mapping function which can apply a functional argument to each "element" of an instance of this type, and which will return NIL unless the result of every such application was non-NIL. *FN* must be a function of two arguments: the aggregate and the function to be applied. For example, the EVERYFN for the built-in type LISTP is EVERY. As described on page 23.27, the Decl package uses the EVERYFN property of the aggregate type to construct a type test for aggregate type expressions. In fact, it is the presence of an EVERYFN property which allows a type to be used as an aggregate type.[16][17]

BINDFN — is used to compute from the initial value supplied for a DLAMBDA or DPROG variable of this type, the value to which the variable will actually be initialized. *FN* must be a function of one argument which will be applied to the initial value,[18] and which should produce another value which is to be used to make the binding. For example, a BINDFN could be used to bind variables of some type so that new

[14] Actually, any property can be attached to a type, and will be available for use by user functions via the function GETDECLTYPEPROP, described below.

[15] Typically, the TESTFN for a type is derived from its type expression, rather than specified explicitly. The ability to specify the TESTFN is provided for those cases where a predicate is available that is much more efficient than that which would be derived from the type expression. For example, the type SMALLP is defined to have the function SMALLP as its TESTFN, rather than (LAMBDA (DATUM) (AND (NUMBERP DATUM) (FIXP DATUM) (SMALLP DATUM))) as would be derived from the subtype structure.

[16] Note that a type's EVERYFN is *not* used in type tests for that type, but only in type tests for types defined by OF expressions which used this type as the aggregate type. For example, EVERY is not used in determining whether some value satisfies the type LISTP.

[17] The Decl package never applies the EVERYFN of a type to a value without first verifying that the value satisfies that type.

[18] For a DPROG binding, *FN* will be applied to no arguments if the initial value is lexically NIL.

bindings are copies of the initial value. Thus, if FLOATP were given the BINDFN FPLUS, any variable declared FLOATP would be initialized with a new floating box, rather than sharing with that of the original initial value.[19]

SETFN is used for performing a SETQ or SETQQ of variables of this type. *FN* is a function of two arguments, the name of the variable, and its new value. A SETFN is typically used to avoid the allocation of storage for intermediate results. Note that the SETFN is *not* the mechanism for the enforcement of type compatibility, which is checked *after* the assignment has taken place. Also note that not all functions which can change values are affected: in particular, SET and SETN are not.

23.7.13.1 Manipulating Named Types

DECLTYPE is a file package type (page 11.1). Thus all of the operations relating to file package types, e.g. GETDEF, PUTDEF, EDITDEF, DELDEF,[20] SHOWDEF, etc., can be performed on decltypes.

The file package command, DECLTYPES, is provided to dump named decltypes symbolically. They will be written as a series of DECLTYPE forms which will specify only those fields which differ from the corresponding field of their supertype(s). If the type depends on any unnamed types, those types will be dumped (as a compound type expression), continuing up the supertype chain until a named type is found. Care should be exercised to ensure that enough of the named type context is dumped to allow the type definition to remain meaningful.

The functions GETDECLTYPEPROP and SETDECLTYPEPROP, defined analogously to the property list functions for atoms, allow the manipulation of the properties of named types. Setting a property to NIL with SETDECLTYPEPROP removes it from the type.

23.7.14 Relations Between Types

The notion of equivalence of two types is not well defined. However, type equivalence is rarely of interest. What is of interest is type *inclusion*, i.e. whether one type is a supertype or subtype of another. The predicate COVERS can be used to determine whether the values of one type include those of another.

(COVERS *HI* *LO*) [Function]

is T if *HI* can be found on some (possibly empty) supertype chain of *LO*; else NIL. Thus, (COVERS 'FIXP (DECLOF 4))=T, even though the DECLTYPE of 4 is SMALLP, not FIXP. The extremal cases are the obvious identities: (COVERS 'ANY *ANYTYPE*) = (COVERS *ANYTYPE* 'NONE) = (COVERS *X* *X*) for any type *X* = T.

COVERS allows declaration based transformations of a form which depend on elements of the form being of a certain type to express their applicability conditions in terms of the weakest type to which they

[19] The BINDFN, if any, associated with a type may be suppressed in a declaration context by creating a subtype with the type expression operator SHARED, as described on page 23.27.

[20] Deleting a named type could possibly invalidate other type definitions that have the named type as a subtype or supertype. Consequently, the deleted type is simply unnamed and left in the type space as long as it is needed.

apply, without explicit concern for other types which may be subtypes of it. For example, if a particular transformation is to be applied whenever an element is of type NUMBERP, the program which applies that transformation does not have to check whether the element is of type SMALLP, LARGEP, FIXP, FLOATP, etc., but can simply ask whether NUMBERP COVERS the type of that element.

The elementary relations among the types, out of which arbitrary traversals of the type space can be constructed, are made available via:

(SUBTYPES *TYPE*) [Function]

Returns the list of types which are *immediate* subtypes of *TYPE*.

(SUPERTYPES *TYPE*) [Function]

Returns the list of types which are *immediate* supertypes of *TYPE*.

23.7.15 The Declaration Database

One of the primary uses of type declarations is to provide information that other systems can use to interpret or optimize code. For example, one might choose to write all arithmetic operations in terms of general functions like PLUS and TIMES and then use variable declarations to substitute more efficient, special purpose code at compile time based on the types of the operands. To this end, a data base of declarations is made available by the Decl package to support these operations.

(DECLOF *FORM*) [Function]

Returns the type of *FORM* in the current declaration context.[21] If *FORM* is an atom, DECLOF will look up that atom directly in its database of current declarations. Otherwise, DECLOF will look on the property list of (CAR *FORM*) for a DECLOF property, as described below. If there is no DECLOF property, DECLOF will check if (CAR *FORM*) is one of a large set of functions of known result type (e.g., the arithmetic functions). Failing that, if (CAR *FORM*) has a MACRO property, DECLOF will apply itself to the result of expanding (with EXPANDMACRO, page 5.19) the macro definition. Finally, if *FORM* is a Lisp program element that DECLOF "understands" (e.g., a COND, PROG, SELECTQ, etc.), DECLOF applies itself recursively to the part(s) of the contained form which will be returned as value.

DECLOF [Property Name]

Allows the specification of the type of the values returned by a particular function. The value of the DECLOF property can be either a type, i.e. a type name or a type expression, or a list of the form (FUNCTION *FN*), where *FN* is a function object. *FN* will be applied (by DECLOF) to the form whose CAR has this DECLOF property on its property list. The value of this function application will then be considered to be the type of the form.

[21] The "current declaration context" is defined by the environment at the time that DECLOF is called. Code reading systems, such as the compiler and the interpreter, keep track of the lexical scope within which they are currently operating, in particular, which declarations are currently in effect. Note that (currently) DECLOF does *not* have access to any global data base of declarations. For example, DECLOF does not have information available about the types of the arguments of, or the value returned by, a particular function, unless it is currently "inside" of that function. However, the DECLOF property (described below) can be used to inform DECLOF of the type of the value returned by a particular function.

As an example of how declarations can be used to automatically generate more efficient code, consider an arithmetic package. Declarations of numeric variables could be used to guide code generation to avoid the inefficiencies of Interlisp's handling of arithmetic values. Not only could the generic arithmetic functions be automatically specialized, as suggested above, but by redefining the BINDFN and the SETFN properties for the types FLOATP and LARGEP to re-use storage in the appropriate contexts (i.e., when the new value can be determined to be of the appropriate type), tremendous economies could be realized by not allocating storage to intermediate results which must later be reclaimed by the garbage collector. The Decl package has been used as the basis for several such code optimizing systems.

23.7.16 Declarations and Masterscope

The Decl package notifies MASTERSCOPE about type declarations and defines a new MASTERSCOPE relation, TYPE, which depends on declarations. Thus, the user can ask questions such as "WHO USES MUMBLE AS A TYPE?," "DOES FOO USE FIXP AS A TYPE?," and so on.

23.8 TRANSOR

Note: TRANSOR is a LispUsers package contained on the file TRANSOR.DCOM.

TRANSOR is a LISP-to-LISP translator intended to help the user who has a program coded in one dialect of LISP and wishes to carry it over to another. The user loads TRANSOR along with a file of transformations. These transformations describe the differences between the two LISPs, expressed in terms of Interlisp editor commands needed to convert the old to new, i.e. to edit forms written in the source dialect to make them suitable for the target dialect. TRANSOR then sweeps through the user's program and applies the edit transformations, producing an object file for the target system. In addition, TRANSOR produces a file of translation notes, which catalogs the major changes made in the code as well as the forms that require further attention by the user. Operationally, therefore, TRANSOR is a facility for conducting massive edits, and may be used for any purpose which that may suggest.

Since the edit transformations are fundamental to this process, let us begin with a definition and some examples. A transformation is a list of edit commands associated with a literal atom, usually a function name. TRANSOR conducts a sweep through the user's code, until it finds a form whose CAR is a literal atom which has a transformation. The sweep then pauses to let the editor execute the list of commands before going on. For example, suppose the order of arguments for the function TCONC must be reversed for the target system. The transformation for TCONC would then be: ((SW 2 3)). When the sweep encounters the form (TCONC X (FOO)), this transformation would be retrieved and executed, converting the expression to (TCONC (FOO) X). Then the sweep would locate the next form, in this case (FOO), and any transformations for FOO would be executed, etc.

Most instances of TCONC would be successfully translated by this transformation. However, if there were no second argument to TCONC, e.g. the form to be translated was (TCONC X), the command (SW 2 3) would cause an error, which TRANSOR would catch. The sweep would go on as before, but a note would appear in the translation listing stating that the transformation for this particular form failed to work. The user would then have to compare the form and the commands, to figure out what caused the problem. One might, however, anticipate this difficulty with a more sophisticated transformation: ((IF (## 3) ((SW 2 3)) ((-2 NIL)))), which tests for a third element and does (SW 2 3) or (-2 NIL) as appropriate. It should be obvious that the translation process is no more sophisticated than the transformations used.

This documentation is divided into two main parts. The first describes how to use TRANSOR assuming that the user already has a complete set of transformations. The second documents TRANSORSET, an interactive routine for building up such sets. TRANSORSET contains commands for writing and editing transformations, saving one's work on a file, testing transformations by translating sample forms, etc.

Two transformations files presently exist for translating programs into Interlisp. <LISP>SDS940.XFORMS is for old BBN LISP (SDS 940) programs, and <LISP>LISP16.XFORMS is for Stanford AI LISP 1.6 programs. A set for LISP 1.5 is planned.

23.8.1 Using TRANSOR

The first and most exasperating problem in carrying a program from one implementation to another is simply to get it to read in. For example, SRI LISP uses / exactly as Interlisp uses %, i.e. as an escape character. The function PRESCAN exists to help with these problems: the user uses PRESCAN to perform an initial scan to dispose of these difficulties, rather than attempting to TRANSOR the foreign sourcefiles directly.

PRESCAN copies a file, performing character-for-character substitutions. It is hand-coded and is much faster than either READC's or text-editors.

(PRESCAN *FILE CHARLST*) [Function]

Makes a new version of *FILE*, performing substitutions according to *CHARLST*. Each element of *CHARLST* must be a dotted pair of two character codes, (*OLD-CHAR-CODE* . *NEW-CHAR-CODE*).

For example, SRI files are PRESCANed with *CHARLST* = ((37 . 47) (47 . 37)), which exchanges slash (47) and percent-sign (37).

The user should also make sure that the treatment of double quotes by the source and target systems is similar. In Interlisp, an unmatched double-quote (unless protected by the escape character) will cause the rest of the file to read in as a string.

Finally, the lack of a STOP at the end of a file is harmless, since TRANSOR will suppress END OF FILE errors and exit normally.

23.8.2 Translating

TRANSOR is the top-level function of the translator itself, and takes one argument, a file to be translated. The file is assumed to contain a sequence of forms, which are read in, translated, and output to a file called {FILE}.TRAN. The translation notes are meanwhile output to {FILE}.LSTRAN. Thus the usual sequence for bring a foreign file to Interlisp is as follows: PRESCAN the file; examine code and transformations, making changes to the transformations if needed; TRANSOR the file; and clean up remaining problems, guided by the notes. The user can now make a pretty file and proceed to exercise and check out his program. To export a file, it is usually best to TRANSOR it, then PRESCAN it, and perform clean-up on the foreign system where the file can be loaded.

(TRANSOR *FILE*) [Function]

Translates *FILE*. Prettyprints translation on {FILE}.TRAN; translation listing on {FILE}.LSTRAN.

(TRANSORFORM *FORM*) [Function]

FORM is a LISP form. Returns the (destructively) translated form. The translation listing is dumped to the primary output file.

(TRANSORFNS *FNLST*) [Function]

FNLST is a list of function names whose interpreted definitions are destructively translated. Listing to primary output file.

TRANSORFORM and TRANSORFNS can be used to translate expressions that are already in core, whereas TRANSOR itself only works on files.

23.8.3 The Translation Notes

The translation notes are a catalog of changes made in the user's code, and of problems which require, or may require, further attention from the user. This catalog consists of two cross-indexed sections: an index of forms and an index of notes. The first tabulates all the notes applicable to any form, whereas the second tabulates all the forms to which any one note applies. Forms appear in the index of forms in the order in which they were encountered, i.e. the order in which they appear on the source and output files. The index of notes shows the name of each note, the entry numbers where it was used, and its text, and is alphabetical by name. The following sample was made by translating a small test file written in SRI LISP.

```
LISTING FROM TRANSORING OF FILE TESTFILE.;7
     DONE ON 1-NOV-71 20:10:47

                                                  INDEX OF FORMS
1. APPLY/EVAL at
  [DEFINEQ
      (FSET (LAMBDA &
            (PROG ...3...
                (SETQ Z (COND
                    ((ATOM (SETQ --))
                     (COND
                       ((ATOM (SETQ Y (NLSETQ "(EVAL W)")))
                        --)
                     --))
                    --))
            --  ]
2. APPLY/EVAL at
  [DEFINEQ
     (FSET (LAMBDA &
           (PROG ...3...
                (SETQ Z (COND
                   ((ATOM (SETQ --))
                     (COND
                   ((ATOM (SETQ --))
                         "(EVAL (NCONS W))")
                        --))
                    --))
            --   ]
3. MACHINE-CODE at
  [DEFINEQ
     (LESS1 (LAMBDA &
            (PROG ...3...
                 (COND
                   ...2...
                   ((NOT (EQUAL (SETQ X2 "(OPENR (MAKNUM & -))"
                               )
                             --))
                    --))
            --    ]
4. MACHINE-CODE at
  [DEFINEQ
     (LESS1 (LAMBDA &
            (PROG ...3...
                 (COND
                   ...2...
                   ((NOT (EQUAL & (SETQ Y2
                               "(OPENR (MAKNUM & --))")))
                    --))
              --    ]
                                                      INDEX OF NOTES
APPLY/EVAL at 1, 2.
     TRANSOR will translate the arguments of the APPLY or EVAL expression, but
the user must make sure that the run-time evaluation of the arguments returns
a BBN-compatible expression.
MACHINE-CODE at 3, 4.
     Expression dependent on machine-code.  User must recode.
```

The translation notes are generated by the transformations used, and therefore reflect the judgment of their author as to what should be included. Straightforward conversions are usually made without comment; for example, the DEFPROPs in this file were quietly changed to DEFINEQs. TRANSOR found four noteworthy forms on the file, and printed an entry for each in the index of forms, consisting of an entry number, the name of the note, and a printout showing the precise location of the form. The form appears in double-quotes and is the last thing printed, except for closing parentheses and dashes. An ampersand represents one non-atomic element not shown, and two or more elements not shown are represented as ...*N*..., where *N* is the number of elements. Note that the printouts describe expressions on the output file rather than the source file; in the example, the DEFPROPs of SRI LISP have been replaced with DEFINEQs.

23.8.4 Errors and Messages

TRANSOR records its progress through the source file by terminal printouts which identify each expression as it is read in. Progress within large expressions, such as a long DEFINEQ, is reported every three minutes by a printout showing the location of the sweep.

If a transformation fails, TRANSOR prints a diagnostic to the teletype which identifies the faulty transformation, and resumes the sweep with the next form. The translation notes will identify the form which caused this failure, and the extent to which the form and its arguments were compromised by the error.

If the transformation for a common function fails repeatedly, the user can type control-H. When the system goes into a break, he can use TRANSORSET to repair the transformation, and even test it out (see TEST command, page 23.36). He may then continue the main translation with OK.

23.8.5 TRANSORSET

To use TRANSORSET, type (TRANSORSET) to Interlisp. TRANSORSET will respond with a + sign, its prompt character, and await input. The user is now in an executive loop which is like EVALQT with some extra context and capabilities intended to facilitate the writing of transformations. TRANSORSET will thus progress APPLY and EVAL input, and execute history commands just as EVALQT would. Edit commands, however, are interpreted as additions to the transformation on which the user is currently working. TRANSORSET always saves on a variable named CURRENTFN the name of the last function whose transformation was altered or examined by the user. CURRENTFN thus represents the function whose transformation is currently being worked on. Whenever edit commands are typed to the + sign, TRANSORSET will add them to the transformation for CURRENTFN. This is the basic mechanism for writing a transformation. In addition, TRANSORSET contains commands for printing out a transformation, editing a transformation, etc., which all assume that the command applies to CURRENTFN if no function is specified. The following example illustrates this process.

```
←TRANSORSET()
+FN TCONC                       [1]
TCONC
+(SW 2 3)                       [2]
+TEST (TCONC A B)               [3]
P
(TCONC B A)
+TEST (TCONC X)                 [4]
TRANSLATION ERROR: FAULTY TRANSFORMATION
TRANSFORMATION: ((SW 2 3))      [5]
OBJECT FORM:         (TCONC X)

1. TRANSFORMATION ERROR AT      [6]
  "(TCONC X)"

(TCONC X)
+(IF (## 3) ((SW 2 3)) ((-2 NIL]     [7]
+SHOW
TCONC
  [(SW 2 3)
   (IF (## 3)                    [8]
       ((SW 2 3))
       ((-2 NIL]
TCONC
```

```
+ERASE                        [9]
TCONC
+REDO IF                      [10]
+SHOW
TCONC
  [(IF (## 3)
       ((SW 2 3))
       ((-2 NIL]
TCONC
+TDST
=TEST                         [11]
(TCONC NIL X)
+
```

In this example, the user begins by using the FN command to set CURRENTFN to TCONC *[1]*. He then adds to the (empty) transformation for TCONC a command to switch the order of the arguments *[2]* and tests the transformation *[3]*. His second TEST *[4]* fails, causing an error diagnostic *[5]* and a translation note *[6]*. He writes a better command *[7]* but forgets that the original SW command is still in the way *[8]*. He therefore deletes the entire transformation *[9]* and redoes the IF *[10]*. This time, the TEST works *[11]*.

23.8.6 TRANSORSET Commands

The following commands for manipulating transformations are all Prog. Asst. commands which treat the rest of their input line as arguments. All are undoable.

FN [Transorset Command]

Resets CURRENTFN to its argument, and returns the new value. In effect FN says you are done with the old function (as least for the moment) and wish to work on another. If the new function already has a transformation, the message (OLD TRANSFORMATIONS) is printed, and any editcommands typed in will be added to the end of the existing commands. FN followed by a carriage return will return the value of CURRENTFN without changing it.

SHOW [Transorset Command]

Command to prettyprint a transformation. SHOW followed by a carriage return will show the transformation for CURRENTFN, and return CURRENTFN as its value. SHOW followed by one or more function names will show each one in turn, reset CURRENTFN to the last one, and return the new value of CURRENTFN.

EDIT [Transorset Command]

Command to edit a transformation. Similar to SHOW except that instead of prettyprinting the transformation, EDIT gives it to EDITE. The user can then work on the transformation until he leaves the editor with OK.

ERASE [Transorset Command]

Command to delete a transformation. Otherwise similar to SHOW.

TEST [Transorset Command]

Command for checking out transformations. TEST takes one argument, a form for translation. The translation notes, if any, are printed to the teletype, but in an abbreviated format which omits the index of notes. The value returned is the translated form. TEST saves a copy of its argument on the free variable TESTFORM, and if no argument is given, it uses TESTFORM, i.e. tries the previous test again.

DUMP [Transorset Command]

Command to save your work on a file. DUMP takes one argument, a filename. The argument is saved on the variable DUMPFILE, so that if no argument is provided, a new version of the previous file will be created.

The DUMP command creates files by MAKEFILE. Normally *FILE*FNS will be unbound, but the user may set it himself; functions called from a transformation by the E command may be saved in this way. DUMP makes sure that the necessary command is included on the *FILE*VARS to save the user's transformations. The user may add anything else to his *FILE*VARS that he wishes. When a transformation file is loaded, all previous transformations are erased unless the variable MERGE is set to T.

EXIT [Transorset Command]

Exits TRANSORSET, returning NIL.

23.8.7 The REMARK Feature

The translation notes are generated by those transformations that are actually executed via an edit macro called REMARK. REMARK takes one argument, the name of a note. When the macro is executed, it saves the appropriate information for the translation notes, and adds one entry to the index of forms. The location that is printed in the index of forms is the editor's location when the REMARK macro is executed.

To write a transformation which makes a new note, one must therefore do two things: define the note, i.e. choose a new name and associate it with the desired text; and call the new note with the REMARK macro, i.e. insert the edit command (REMARK *NAME*) in some transformation. The NOTE command, described below, is used to define a new note. The call to the note may be added to a transformation like any other edit command. Once a note is defined, it may be called from as many different transformations as desired.

The user can also specify a remark with a new text, without bothering to think of a name and perform a separate defining operation, by calling REMARK with more than one argument, e.g. (REMARK *TEXT-OF-REMARK*). This is interpreted to mean that the arguments are the text. TRANSORSET notices all such expressions as they are typed in, and handles naming automatically; a new name is generated[22] and defined with the text provided, and the expression itself is edited to be (REMARK *GENERATED-NAME*). The following example illustrates the use of REMARK.

```
←TRANSORSET()
+NOTE GREATERP/LESSP (BBN'S GREATERP AND LESSP ONLY TAKE TWO ARGUMENTS, WHEREAS
SRI'S FUNCTIONS TAKE AN INDEFINITE NUMBER. AT THE PLACES NOTED HERE, THE SRI
CODE USED MORE THAN TWO ARGUMENTS, AND THE USER MUST RECODE.]  [1]
```

[22] The name generated is the value of CURRENTFN suffixed with a colon, or with a number and a colon.

```
GREATERP/LESSP
+FN GREATERP
GREATERP
+(IF (IGREATERP (LENGTH (##))3) NIL ((REMARK GREATERP/LESSP]   [2]
+FN LESSP
LESSP
+REDO IF   [3]
+SHOW
LESSP
  [(IF (IGREATERP (LENGTH (##))
                 3)
       NIL
       ((REMARK GREATERP/LESSP]
LESSP
+FN ASCII
(OLD TRANSFORMATIONS)
ASCII
+(REMARK ALTHOUGH THE SRI FUNCTION ASCII IS IDENTICAL TO THE BBN FUNCTION CHARACTER
THE USER MUST MAKE SURE THAT THE CHARACTER BEING CREATED SERVES THE SAME PURPOSE
ON BOTH SYSTEMS, SINCE THE CONTROL CHARACTERS ARE ALL ASSIGNED DIFFRENTLY.]   [4]

+SHOW   [5]
ASCII
  ((1 CHARACTER)
   (REMARK ASCII:))
ASCII
+NOTE ASCII:   [6]
EDIT
*NTH -2
*P
... ASSIGNED DIFFRENTLY.)
*(2 DIFFERENTLY.)
OK
ASCII:
+
```

In this example, the user defines a note named GREATERP/LESSP by using the NOTE command *[1]*, and writes transformations which call this note whenever the sweep encounters a GREATERP or LESSP with more than two arguments *[2]* and *[3]*. Next, the implicit naming feature is used *[4]* to add a REMARK command to the transformation for ASCII, which has already been partly written. The user realizes he mistyped part of the text, so he uses the SHOW command to find the name chosen for the note *[5]*. Then he uses the NOTE command on this name, ASCII:, to edit the note *[6]*.

NOTE [Transorset Command]

First argument is note name and must be a literal atom. If already defined, NOTE edits the old text; otherwise it defines the name, reading the text either from the rest of the input line or from the next line. The text may be given as a line or as a list. Value is name of note.

The text is actually stored.[23] as a comment, i.e. a * and %% are added in front when the note is first defined. The text will therefore be lower-cased the first time the user DUMPs (see page 6.52).

DELNOTE [Transorset Command]

Deletes a note completely (although any calls to it remain in the transformations).

23.8.8 Controlling the Sweep

TRANSOR's sweep searches in print-order until it finds a form for which a transformation exists. The location is marked, and the transformation is executed. The sweep then takes over again, beginning from the marked location, no matter where the last command of the transformation left the editor. User transformations can therefore move around freely to examine the context, without worrying about confusing the translator. However, there are many cases where the user wants his transformation to guide the sweep, usually in order to direct the processing of special forms and FEXPRs. For example, the transformation for QUOTE has only one objective: to tell the sweep to skip over the argument to QUOTE, which is (presumably) not a LISP form. NLAM is an edit macro that permits this.

NLAM [Transorset Command]

An atomic edit macro which sets a flag which causes the sweep to skip the arguments of the current form when the sweep resumes.

Special forms such as COND, PROG, SELECTQ, etc., present a more difficult problem. For example, (COND (A B)) is processed just like (FOO (A B)): i.e. after the transformation for COND finishes, the sweep will locate the "next form," (A B), retrieve the transformation for the function A, if any, and execute it. Therefore, special forms must have transformations that preempt the sweep and direct the translation themselves. The following two atomic edit macros permit such transformations to process their forms, translating or skipping over arbitrary subexpressions as desired.

DOTHIS [Transorset Command]

Translates the editor's current expression, treating it as a single form.

DOTHESE [Transorset Command]

Translates the editor's current expression, treating it as a list of forms.

For example, a transformation for SETQ might be (3 DOTHIS).[24] This translates the second argument to a SETQ without translating the first. For COND, one might write (1 (LPQ NX DOTHESE)), which locates each clause of the COND in turn, and translates it as a list of forms, instead of as a single form.

The user who is starting a completely new set of transformations must begin by writing transformations for all the special forms. To assist him in this and prevent oversights, the file <LISP>SPECIAL.XFORMS contains a set of transformations for LISP special forms, as well as some other transformations which should also be included. The user will probably have to revise these transformations substantially, since they merely perform sweep control for Interlisp, i.e. they make no changes in the object code. They are provided chiefly as a checklist and tutorial device, since these transformations are both the first to be written and the most difficult, especially for users new to the Interlisp editor.

[23] On the global list USERNOTES.

[24] Recall that a transformation is a list of edit commands. In this case, there are two commands, 3 and DOTHIS.

When the sweep mechanism encounters a form which is not a list, or a form CAR of which is not an atom, it retrieves one of the following special transformations.

NLISTPCOMS [Variable]

Global value is used as a transformation for any form which is not a list.

For example, if the user wished to make sure that all strings were quoted, he might set NLISTPCOMS to ((IF (STRINGP (##)) ((ORR ((← QUOTE))((MBD QUOTE)))) NIL)).

LAMBDACOMS [Variable]

Global value is used as a transformation for any form, CAR of which is not an atom.

These variables are initialized by <LISP>SPECIAL.XFORMS and are saved by the DUMP command. NLISTPCOMS is initially NIL, making it a NO-OP. LAMBDACOMS is initialized to check first for open LAMBDA expressions, processing them without translation notes unless the expression is badly formed. Any other forms with a non-atomic CAR are simply treated as lists of forms and are always mentioned in the translation notes. The user can change or add to this algorithm simply by editing or resetting LAMBDACOMS.

23.9 WHEREIS PACKAGE

Note: The WHEREIS is a LispUsers package that is contained on the file WHEREIS.COM. *WHEREIS requires the hash file package (page 23.41). Loading* WHEREIS.COM *will also load* HASH.COM, *if it has not already been loaded.*

This package extends the function WHEREIS (page 11.10) such that, when asked about a given name as a function, WHEREIS will consult not only the commands of files that have been noticed by the file package (page 11.1) but also a hashfile database (page 23.41) that associates function names with filenames.

(WHEREIS *NAME TYPE FILES FN*) [Function]

Behaves exactly like the definition on page 11.10 unless *TYPE*=FNS (or NIL) and *FILES*=T. In this case, WHEREIS will consult, in addition to the files on FILELST, the hashfile that is the value of WHEREIS.HASH (initially <LISPUSER>WHEREIS.HASH).

Note: Most system functions call WHEREIS with *FILES*=T, so loading this package automatically makes the information contained in the WHEREIS database available throughout the system.

Information may be added to a WHEREIS hashfile by explicitly calling the following function:

(WHEREISNOTICE *FILEGROUP NEWFLG*) [Function]

Inserts the information about all of the functions on the files in *FILEGROUP* into the WHEREIS data base contained on (the value of) WHEREIS.HASH. *FILEGROUP* is given as a filegroup argument to DIRECTORY (page 14.6), so &, $, etc. may be used. If *NEWFLG*=T, a new version of WHEREIS.HASH will be created containing the database for the functions specified in *FILEGROUP*.

23.10 HASH FILES

Note: The hash file facility is a LispUsers package that is contained on the file HASH.COM. *It currently only works in Interlisp-10.*

The hash file facility permits information associated with string or atom "keys" to be stored on and retrieved from files. The information (or "values") associated with the keys in a file may be numbers, strings, or arbitary Interlisp expressions. The associations are maintained by a hashing scheme that minimizes the number of page-maps it takes to access a value from its key.

A hashfile may contain information other than key-value associations. The user may print on the file using ordinary printing functions (e.g. PRIN1, PRINTDEF), and he may also store non-character information (e.g. binary data) formatted to suit his particular applications. This information is stored in regions of the file distinct from the hash index. The hash index can be used to locate non-hash information, if the necessary file addresses are stored as hash values.

A hashfile is created by the function CREATEHASHFILE:

(CREATEHASHFILE *FILE VALUETYPE ITEMLENGTH #ENTRIES*) [Function]

A new version of *FILE* is opened and initialized as a hashfile. *VALUETYPE* is an atom interpreted as follows:

NUMBER The values are 24-bit unsigned integers.

STRING The values are strings with less than 128 characters.

EXPR The values are arbitrary Interlisp expressions. The values are stored by printing them in the file with readtable HASHFILERDTBL, initially ORIG.

SMALLEXPR

The values are arbitrary Interlisp expressions such that (NCHARS *VALUE* T HASHFILERDTBL) is less than 128. Storing and retrieving is more efficient than if *VALUETYPE*=EXPR .

SYMBOLTABLE

The values are 24-bit unsigned integers, as when *VALUETYPE*=NUMBER, except that the numbers are treated as the addresses of "symbols" located on non-hash pages in the file. See the discussion of symbol-tables below.

The other arguments to CREATEHASHFILE are optional. *ITEMLENGTH* is the user's estimate of the average number of characters in the entries he expects to store in the hashfile (= the average key length plus the average number of characters in the values for *VALUETYPE* STRING or SMALLEXPR). *#ENTRIES* is an estimate of the the total number of key-value associations he is likely to store. These two arguments determine how many pages in the file will be initially allocated as hash-pages; accurate estimates can reduce the number of times that the file must be rehashed as information is stored in it. If these arguments are not given, reasonable defaults are supplied.

After being initialized, *FILE* is left open and CREATEHASHFILE returns as its value a "hashfile datum," a handle on the hashfile that may be used as an argument for most of the functions described below.

(OPENHASHFILE *FILE ACCESS*) [Function]

Re-opens the previously existing hashfile *FILE*. *ACCESS* may be INPUT (or NIL), in which case *FILE* is opened for reading only, or BOTH, in which case *FILE* is open for both input and output. Causes an error NOT A HASHFILE, if *FILE* is not recognized as a hashfile.

If *ACCESS* is BOTH and *FILE* is a hashfile open for reading only, OPENHASHFILE attempts to close it and re-open it for writing. Otherwise, if *FILE* designates an already open hashfile, OPENHASHFILE is a no-op.

OPENHASHFILE returns a hashfile datum.

(HASHFILEP *X*) [Function]

Returns *X* if *X* is a hashfile datum (i.e., a value returned by CREATEHASHFILE or OPENHASHFILE). If *X* is NIL, returns SYSHASHFILE if it is a hashfile datum. If *X* is the name of an open hashfile, returns the corresponding hashfile datum. Otherwise, returns NIL.

The following functions require an open hashfile as an argument, i.e. an object for which HASHFILEP is non-NIL.

(PUTHASHFILE *KEY VALUE HASHFILE*) [Function]

Puts *VALUE* in *HASHFILE*, indexed under *KEY*. If *VALUE* is NIL, any previous entry for *KEY* is deleted.

(GETHASHFILE *KEY HASHFILE*) [Function]

Returns the value corresponding to *KEY* in *HASHFILE*. For files where *VALUETYPE* is STRING, NUMBER, or SYMBOLTABLE, the value returned by GETHASHFILE is temporary in that any subsequent calls to a hashfile or page mapping function may smash it. CONCAT or MKATOM must be applied if the value is a string, or IPLUS if it is a number, in order to make the value permanent.

(HASHFILEPROP *HASHFILE PROP*) [Function]

Returns the value of the *PROP* property of *HASHFILE*. The recognized *PROP*s and the values returned are:

VALUETYPE
One of NUMBER, STRING, EXPR, SMALLEXPR, or SYMBOLTABLE.

NAME The full name of the file.

ACCESS BOTH if file is open for writing, INPUT if it is read-only.

(HASHFILENAME *HASHFILE*) [Function]

Same as (HASHFILEPROP *HASHFILE* 'NAME).

(CLOSEHASHFILE *HASHFILE*) [Function]

Same as (CLOSEF (HASHFILEPROP *HASHFILE* 'NAME)).

The function HASHSTATUS can be used as a STATUS function for WHENCLOSE (page 6.11) to restore the state of a hashfile when a SYSOUT is resumed. If HASHSTATUS is used, the PERMSTATUS package (page 23.17) must also be loaded.

(MAPHASHFILE *HASHFILE MAPFN*) [Function]

For each entry in *HASHFILE*, performs (*MAPFN KEY* (GETHASHFILE *KEY HASHFILE*)). If *MAPFN* is a function of only one argument, performs (*MAPFN KEY*) thereby avoiding the call to GETHASHFILE needed to obtain the value. *KEY* is temporary, as for GETHASHFILE. *VALUE* is also temporary, for STRING, NUMBER, and SYMBOLTABLE files.

(REHASHFILE *HASHFILE*) [Function]

After many insertions and deletions much of the space in a hashfile may be unusable. REHASHFILE reclaims that space by rehashing all the keys. The information on non-hash pages in the file is not altered or moved, except that the print name pointers in a SYMBOLTABLE file are updated (see below).

(COPYHASHFILE *HASHFILE NEWNAME FN VTYPE*) [Function]

Calls CREATEHASHFILE to open *NEWNAME* as a hashfile, with *VALUETYPE*, *ITEMLENGTH* and *#ENTRIES* determined by examining the open hashfile *HASHFILE*. Then maps through all the keys in *HASHFILE*, doing the equivalent of:

```
(PUTHASHFILE
   KEY
   (GETHASHFILE KEY HASHFILE)
   NEWHASHFILE)
```

for each key *KEY*. In essence, COPYHASHFILE copies the hash portion of *HASHFILE* to *NEWNAME*.

If *FN* is given, then it is applied to the successive values of *HASHFILE*, the old *HASHFILE*, and the new hashfile, and the value returned is used as the value in the new file. In effect,

```
(PUTHASHFILE
   KEY
   (FN (GETHASHFILE KEY HASHFILE)
        HASHFILE
        NEWHASHFILE)
   NEWHASHFILE)
```

is evaluated for each key. Thus, the user can intervene as each key is processed in order to copy information associated with the key that resides on non-hash pages.

For example, an EXPR file could be implemented by printing the full expressions in a NUMBER file's printing region (see below) and storing their byte-positions as hash values. Instead of reading an expression into internal data structures before writing it out to the new file, a *FN* could be given that transferred the expression to the new file more efficiently, via COPYBYTES. The function would return the byte-position on the new file where the expression ended up. (Actually, this is the way EXPR files are copied if *FN* is not specified.)

If *FN* is given, then *VTYPE*, if specified, is a temporary valuetype (NUMBER, STRING, etc.) to be used during copying. This permits the user to force the valuetype of both files to one more suited for *FN*, e.g. SMALLEXPR to STRING or EXPR to NUMBER, as in the example. *VTYPE* does not affect the permanent valuetype of either file.

(HASHFILESPLST *HASHFILE*) [Function]

Returns a "generator" for the keys in *HASHFILE* that is acceptable as an argument to FIXSPELL (page 15.18). Thus, (FIXSPELL BADWORD 70 (HASHFILESPLST *HASHFILE*)) will spelling correct a word using the keys in *HASHFILE*.

(LOOKUPHASHFILE *KEY VALUE HASHFILE CALLTYPE*) [Function]

A generalized entry for inserting and retrieving values; provides certain options not available with GETHASHFILE or PUTHASHFILE. LOOKUPHASHFILE looks up *KEY* in *HASHFILE*. *CALLTYPE* is an atom or a list of atoms. These keywords are interpreted as follows:

RETRIEVE

If *KEY* is found, then if *CALLTYPE* is or contains RETRIEVE, the old value is returned from LOOKUPHASHFILE; otherwise returns T.

DELETE If *CALLTYPE* is or contains DELETE, the value associated with *KEY* is deleted from the file.

REPLACE If *CALLTYPE* is or contains REPLACE, the old value is replaced with *VALUE*.

INSERT If *CALLTYPE* is or contains INSERT, LOOKUPHASHFILE inserts value as the value associated with *KEY*.

If *KEY* is not found, LOOKUPHASHFILE returns NIL.

Examples:

To either return an old value or insert a new value in the file if one does not already exist, perform (LOOKUPHASHFILE *KEY NEWVALUE HASHFILE* '(INSERT RETRIEVE)). The value returned will be NIL if *NEWVALUE* was inserted, or the old value if *KEY* was found.

To merely check whether *KEY* exists in the file without actually retrieving its value (which may be expensive for the more general valuetypes), perform (LOOKUPHASHFILE *KEY* NIL *HASHFILE* NIL).

The function PUTHASHFILE is defined as:

```
(LAMBDA (KEY VALUE HASHFILE)
    (if VALUE=NIL
      then (LOOKUPHASHFILE KEY NIL HASHFILE 'DELETE)
      else (LOOKUPHASHFILE KEY VALUE HASHFILE '(INSERT REPLACE))
           VALUE))
```

And GETHASHFILE is defined as:

```
(LAMBDA (KEY HASHFILE)
    (LOOKUPHASHFILE KEY NIL HASHFILE 'RETRIEVE))
```

23.10.1 Unstructured Pages and Symbol Tables

The non-hash information in a hash-file may be formatted as printed character strings or binary data. Printed information resides in a file's "printing region", while binary data is stored on "unstructured pages".

Unstructured pages in a file are allocated and deallocated by the hash package so that they do not encroach on hash or printing pages. Other than that, the user has complete freedom to map them in for arbitrary reading and writing. The primitive operations are:

(GETPAGE *HASHFILE N*) [Function]

Returns the page number of a free page in *HASHFILE*. If *N* is given, then the user is guaranteed that the page returned is the first of *N* contiguous pages all of which are free.

(DELPAGE *PAGE# HASHFILE*) [Function]

Removes page *PAGE#* from *HASHFILE*. *PAGE#* should be the number of an unstructured page, either a value of GETPAGE or within the block of free pages guaranteed by GETPAGE. The contents of the page in the file are lost, and the page itself becomes available for re-allocation either by GETPAGE or internally as a hash page. If *PAGE#* happens to be the number of a hash page, the hashing information will be destroyed.

Unstructured pages are available on hashfiles so that the user can link hash keys to data in special formats. For example, the user might associate lists of properties with a key by writing the properties on an unstructured page, and then storing the file address of the properties as the value of the key in a NUMBER file.

A SYMBOLTABLE hashfile provides an additional feature that makes it possible to implement arbitrary file-resident symbol processing systems. The user may store the data to be associated with a key on unstructured pages, and he can then link the file address to the key via PUTHASHFILE, as described above. The difference between a NUMBER and SYMBOLTABLE file is that for a SYMBOLTABLE, the hash package also stores the reverse link from the file address to the key. This makes it possible to obtain a "print-name" for an address on an unstructured page, via the function GETPNAME:

(GETPNAME *FILEADR HASHFILE*) [Function]

Returns a temporary string containing the characters of the key whose hash value is the 24-bit unsigned *FILEADR*. Causes an error if *HASHFILE* is not a SYMBOLTABLE file.

The hash package automatically updates the print-name information for the file address if the key is relocated by rehashing, and it destroys the back-link if the value for the key is deleted. A SYMBOLTABLE file imposes one restriction on the way unstructured pages are treated: If a file address is stored as a hash-value for some key, then the right-most 24 bits of the word at that location in the file are reserved for the use of the hash mechanism.[25] The user must not write into it.

With these primitives, a list-processing system with a 24-bit non-resident address space is easy to build. The user is responsible for allocating "atoms" on unstructured pages, and updating the "atom hash table"

[25] The left-most 12 bits are available and can be used for a number of applications, e.g. to store type-bits.

with PUTHASHFILE. The second (and subsequent) words after an atom address may be used to store the atom's "property list", containing other atom addresses, or other addresses interpreted as pointers to "cons" cells. These can also be allocated on unstructured pages. It is a simple matter to implement the equivalent of CAR, CDR, RPLACA, and RPLACD.

23.10.2 The Printing Region

Hashfiles are organized so that it is always permissible to print at the end of the file with ordinary Interlisp output functions. That is, the file is arranged so that the hash and unstructured pages are always located before the end-of-file for sequential reading and writing. This is accomplished by creating the file with the end-of-file some number of free pages past the last hash or unstructured page. When all free pages below the end-of-file have been used, the end-of-file is moved so that there are again a reservoir of free pages before it.

Thus, the printing region may shift as a result of calls to GETPAGE or PUTHASHFILE, and the user cannot rely on the output from two different printing operations being located at adjacent positions in the file. The expressions printed cannot be retrieved by successive calls to standard reading functions. Instead, the user should record the byte position of each printed expression as a hash value or on an unstructured page so that he may use SETFILEPTR to position the file properly. If he does change the file's byte-pointer, he must be sure to reset it to the end-of-file (e.g. (SETFILEPTR *FILE* -1)) before more printing is done.

23.11 EDITA

Note: EDITA is a LispUsers package contained on the file EDITA.COM. *That portion of EDITA relating to compiled code may not be available in implementations of Interlisp other than Interlisp-10. EDITA also has a* FILEDEF *property so that the user can simply call* EDITA *and the file will be automatically loaded.*

EDITA is an editor for arrays. However, its most frequent application is in editing compiled functions (which are also arrays in Interlisp-10), and a great deal of effort in implementing EDITA, and most of its special features, are in this area. For example, EDITA knows the format and conventions of Interlisp-10 compiled code, and so, in addition to decoding instructions a la DDT (one of the oldest debugging systems still around), EDITA can fill in the appropriate COREVALS, symbolic names for index registers, references to literals, linked function calls, etc. The following output shows a sequence of instructions in a compiled function first as they would be printed by DDT, and second by EDITA.

```
466716/  PUSH 16,LISP&KNIL          3/   PUSH PP,KNIL
466717/  PUSH 16,LISP&KNIL          4/   PUSH PP,KNIL
466720/  HRRZ 1,-12(16)             5/   HRRZ 1,-10(PP)
466721/  CAME 1,LISP&KNIL           6/   CAME 1,KNIL
466722/  JRST 466724                7/   JRST 9[26]
466723/  HRRZ 1,@467575             8/   HRRZ 1,@'BRKFILE
466724/  PUSH 16,1                  9/   PUSH PP,1
466725/  LISP&IOFIL,,467576         10/  PBIND 'BRKZ
466726/  -3,,-3                     11/  -524291
466727/  HRRZ 1,-14(16)             12/  HRRZ 1,-12(PP)
466730/  CAMN 1,467601              13/  CAMN 1,'OK
466731/  JRST 466734                14/  JRST 17
466732/  CAME 1,467602              15/  CAME 1,'STOP
466733/  JRST 466740                16/  JRST 21
466734/  PUSH 16,467603             17/  PUSH PP,'BREAK1
466735/  PUSH 16,467604             18/  PUSH PP,'(ERROR!)
466736/  LISP&FILEN,,467605         19/  CCALL 2,'RETEVAL
466737/  JRST 467561                20/  JRST 422
466740/  CAME 1,467606              21/  CAME 1,'GO
466741/  JRST 466754                22/  JRST 33
466742/  HRRZ 1,@-12(16)            23/  HRRZ 1,@-10(PP)
466743/  PUSH 16,1                  24/  PUSH PP,1
```

Therefore, rather than presenting EDITA as an array editor with some extensions for editing compiled code, we prefer to consider it as a facility for editing compiled code, and point out that it can also be used for editing arbitrary arrays.

23.11.1 Overview

EDITA is envoked by calling the function EDITA:

(EDITA *FN* *COMS*) [Function]

Envokes EDITA to edit the function *FN*. To the user, EDITA looks very much like DDT with Interlisp-10 extensions. If *COMS* is given, it should be a list of commands for EDITA. These are then executed exactly as though they had been typed. EDITA can be exited with the command OK.

Individual registers or cells in the function may be examined by typing their address followed by a slash, e.g.

[26] Note that EDITA prints the addresses of cells contained *in* the function relative to the origin of the function.

```
6/  HRRZ 1,-10(PP)
```

The slash is really a command to EDITA to open the indicated register.[27] Only one register at a time can be open, and only open registers can be changed. To change the contents of a register, the user first opens it, types the new contents, and then closes the register with a carriage-return,[28] e.g.

```
7/  CAME 1,'↑    CAMN 1,'↑cr
```

If the user closes a register without specifying the new contents, the contents are left unchanged. Similarly, if an error occurs or the user types control-E, the open register, if any, is closed without being changed.

23.11.2 Input Protocol

EDITA processes all inputs not recognized as commands in the same way. If the input is the name of an instruction (i.e., an atom with a numeric OPD property), the corresponding number is added to the input value being assembled,[29] and a flag is set which specifies that the input context is that of an instruction.

The general form of a machine instruction is (*OPCODE AC* , @ *ADDRESS* (*INDEX*)) as described on page 22.15. Therefore, in instruction context, EDITA evaluates all atoms (if the atom has a COREVAL property, the value of the COREVAL is used), and then if the atom corresponds to an *AC*,[30] shifts it left 23 bits and adds it to the input value, otherwise adds it directly to the input value, but performs the arithmetic in the low 18 bits.[31] Lists are interpreted as specifying index registers, and the value of CAR of the list (again COREVALs are permitted) is shifted left 18 bits. Examples:

```
PUSH PP, KNIL
HRRZ 1,-10(PP)
CAME 1, 'GO
JRST 33 ORG
```

EDITA cannot in general know whether an address field in an instruction that is typed in is relative or absolute. Therefore, the user must add ORG, the origin of the function, to the address field himself. Note that EDITA would *print* this instruction, JRST 53 ORG, as JRST 53.

The user can also specify the address of a literal via the ' command, see page 23.50. For example, if the literal " UNBROKEN" is in cell 85672, HRRZ 1,'" UNBROKEN" is equivalent to HRRZ 1, 85672.

[27] EDITA also converts absolute addresses of cells within the function to relative address on input. Thus, if the definition of FOO begins at 85660, typing 6/ is *exactly* the same as typing 85666/.

[28] Since carriage-return has a special meaning, EDITA indicates the balancing of parentheses by typing a space.

[29] The input value is initially 0.

[30] i.e., if a "," has not been seen, *and* the value of the atom is less than 16, *and* the low 18 bits of the input value are all zero.

[31] If the absolute value of the atom is greater than 1000000Q, full word arithmetic is used. For example, the indirect bit is handled by simply binding @ to 20000000Q.

When the input context is *not* that of an instruction, i.e., no OPD has been seen, all inputs are evaluated (the value of an atom with a COREVAL property is the COREVAL.) Then numeric values are simply added to the previous input value; non-numeric values *become* the input value.[32]

The only exception to the entire procedure occurs when a register is open that is in the pointer region of the function, i.e., literal table. In this case, atomic inputs are *not* evaluated. For example, the user can change the literal FOO to FIE by simply opening that register and then typing FIE followed by carriage-return, e.g.

```
'FOO/    FOO      FIE^cr
```

Note that this is equivalent to

```
'FOO/    FOO      (QUOTE FIE)^cr
```

23.11.3 EDITA Commands and Variables

^{cr} (carriage-return)

If a register is open and an input was typed, store the input in the register and close it.[33]

If a register is open and nothing was typed, close the register without changing it.

If a register is not open and input was typed, type its value.

ORG

Has the value of the address of the first instruction in the function. i.e., LOC of GETD of the function.

/

Opens the register specified by the low 18 bits of the quantity to the left of the /, and types its contents. If nothing has been typed, it uses the last thing typed by EDITA, e.g.,

```
35/  JRST 53   /   CAME 1,'RETURN   /   RETURN
```

If a register was open, / closes it without changing its contents.

After a / command, EDITA returns to that state of no input having been typed.

tab (control-I)

Same as carriage-return, followed by the address of the quantity to the left of the tab, e.g.,

```
35/   JRST 53 <tab>
53/   CAME 1,'RETURN
```

Note that if a register was open and input was typed, tab will change the open register before closing it, e.g.,

[32] Presumably there is only one input in this case.

[33] If the register is in the unboxed region of the function, the unboxed value is stored in the register.

```
35/    JRST 53    JRST 54 TAB
54/    JRST 70$^{cr}$
35/    JRST 54
```

. (period) Has the value of the address of the current (last) register examined.

line-feed Same as carriage-return followed by `(ADD1 .)/` i.e. closes any open register and opens the *next* register.

↑ Same as carriage-return followed by `(SUB1 .)/`

$Q (<esc>Q) Has as its value the last quantity typed by EDITA e.g.

```
35/   JRST 53    $Q 1$^{cr}$
./    JRST 54
```

`LITS` Has as value the (relative) address of the first literal.

`BOXED` Same as `LITS`

$ (dollar) Has as value the relative address of the last literal in the function.

= Sets `RADIX` (page 6.19) to -8 and types the quantity to the left of the = sign, i.e., if anything has been typed, types the input value, otherwise, types $Q, e.g.

```
35/    JRST 54 =254000241541Q
JRST 54=254000000066Q
```

Following =, `RADIX` is restored and EDITA returns to the no input state.

`OK` Exits EDITA.

? Returns to "no input" state. ? is a "weak" control-E, i.e., it negates any input typed, but does not close any registers.

ADDRESS$_1$, *ADDRESS*$_2$/

Prints the contents of registers *ADDRESS*$_1$ through *ADDRESS*$_2$. . is set to *ADDRESS*$_2$ after the completion.

Output goes to `FILE`, initially set to `T`. The user can also set `FILE` (while in EDITA) to the name of a disc file to redirect the output. (The user is responsible for opening and closing `FILE`.) Note that `FILE` only affects output for the *ADDRESS*$_1$, *ADDRESS*$_2$/ command.

'*x* Corresponds to the ' in LAP. The next expression is read, and if it is a small number, the appropriate offset is added to it. Otherwise, the literal table is searched for *x*, and the value of '*x* is the (absolute) address of that cell. An error is generated if the literal is not found, i.e., ' cannot be used to *create* literals.

:*ATOM* Defines *ATOM* to an address: (1) the value of `$Q` if a register is open, (2) the input if any input was typed, otherwise (3) the value of "`.`" (Only the low 18 bits are used and converted to a relative address whenever possible). For example:

```
35/    JRST 54     :FOO$^{cr}$
```

```
:FIEcr
FIE/  JRST FOO .=35
```

EDITA keeps its symbol tables on two free variables, USERSYMS and SYMLST. USERSYMS is a list of elements of the form (*NAME* . *VALUE*) and is used for *encoding* input, i.e., all variables on USERSYMS are bound to their corresponding values during evaluation of any expression inside EDITA. SYMLST is a list of elements of the form (*VALUE* . *NAME*) and is used for *decoding* addresses. USERSYMS is initially NIL, while SYMLST is set to a list of all the COREVALS. Since the : command adds the appropriate information to both these two lists, new definitions will remain in effect even if the user exits from EDITA and then reenters it later.

Note that the user can effectively define symbols without using the : command by appropriately binding USERSYMS and/or SYMLST before calling EDITA. Also, he can thus use different symbol tables for different applications.

$W (<esc>W)   Search command.

Searching consists of comparing the object of the search with the contents of each register, and printing those that match, e.g.,

```
HRRZ @ $Wcr
8/   HRRZ 1,@'BRKFILE
23/  HRRZ 1,@-10(PP)
28/ HRRZ 1,@-12(PP)
```

The $W command can be used to search either the unboxed portion of a function, i.e., instructions, or the pointer region, i.e., literals, depending on whether or not the object of the search is a number. If any input was typed before the $W, it will be the object of the search, otherwise the next expression is read and used as the object.[34] The user can specify a starting point for the search by typing an address followed by a "," before calling $W, e.g., 1, JRST $W. If no starting point is specified, the search will begin at 0 if the object is a number, otherwise at LITS, the address of the first literal.[35] After the search is completed, "." is set to the address of the last register that matched.

If the search is operating in the unboxed portion of the function, only those fields (i.e., *INSTRUCTION*, *AC*, *INDIRECT*, *INDEX*, and *ADDRESS*) of the object that contain one bits are compared.[36] For example, HRRZ @ $W will find all instances of HRRZ indirect, regardless of *AC*, *INDEX*, and *ADDRESS* fields. Similarly, 'PRINT $W will find all instructions that reference the literal PRINT.[37]

[34] Note that inputs typed before the $W will have been processed according to the input protocol, i.e., evaluated; inputs typed after the $W will not. Therefore, the latter form is usually used to specify searching the literals, e.g., $W FOO is equivalent to (QUOTE FOO) $W.

[35] Thus the only way the user can search the pointer region for a number is to specify the starting point via ",".

[36] Alternately, the user can specify his own mask by setting the variable MASK (while in EDITA), to the appropriate bit pattern.

[37] The user may need to establish instruction context for input without giving a specific instruction. For example, suppose the user wants to find all instructions with *AC*=1 and *INDEX*=PP. In this case, the user can give & as a pseudo-instruction, e.g., type & 1, (PP).

If the search is operating in the pointer region, a "match" is as defined in the editor. For example, $W (&) will find all registers that contain a list consisting of a single expression.

$C (<esc>C) Like $W except only prints the first match, then prints the number of matches when the search finishes.

23.11.4 Editing Arrays

EDITA is called to edit a function by giving it the name of the function. EDITA can also be called to edit an array by giving it the array as its first argument,[38] in which case the following differences are to be noted:

1. decoding - The contents of registers in the unboxed region are boxed and printed as numbers, i.e., they are never interpreted as instructions, as when editing a function.

2. addressing convention - Whereas 0 corresponds to the first instruction of a function, the first element of an array by convention is element number 1.

3. input protocols - If a register is open, lists are evaluated, atoms are not evaluated (except for $Q which is always evaluated). If no register is open, all inputs are evaluated, and if the value is a number, it is added to the "input value".

4. left half - If the left half of an element in the pointer region of an array is not all 0's or NIL, it is printed followed by a ";", e.g.

```
10/   (A B) ; T
```

Similarly, if a register is closed, either its left half, right half, or both halves can be changed, depending on the presence or absence, and position of the ";" e.g.

```
10/  (A B) ; T     B;cr        [changes left]
./   B ; T         NILcr        [changes right]
./   B ; NIL       A ; Ccr     [changes both]
./   A ; C
```

If ";" is used in the unboxed portion of an array, an error will be generated.

[38] the array itself, *not* a variable whose value is an array, e.g., (EDITA FOO), not (EDITA 'FOO).

The `$W` command will look at both halves of elements in the pointer region, and match if either half matches. Note that `$W A ; B` is not allowed.

23.12 CJSYS

Note: Cjsys is a LispUsers package that is contained on the file `CJSYS.COM`. *It only works with Interlisp-10.*

This package provides assistance to Interlisp-10 users who wish to make direct calls on the operating system (via JSYSes). It also makes the coding of certain common `ASSEMBLE` constructions more convenient. The package defines the following functions:

`(JS` *JSYSNAME AC1 AC2 AC3 RESULT*`)` [NLambda Function]

All arguments are evaluated except for *JSYSNAME*. Like `JSYS` (see page 22.6), loads the unboxed values of *AC1*, *AC2*, and *AC3* into the appropriate registers, and executes the JSYS *JSYSNAME*. `JS` differs from `JSYS` in that the JSYS may be indicated by its *symbolic* name, not just by its number. `JS` also generates slightly cleaner code than `JSYS`. `JS` also differs from `JSYS` in that:

(a) if any argument is supplied as `NIL`, then it is not loaded at all, i.e. the corresponding `AC` will contain garbage. (`JSYS` loads the `AC` with 0.)

(b) if *RESULT* is `NIL`, then no value is loaded (interpreted, `JS` returns the string `"garbage result from JS"`).

(c) *RESULT* can be `T`, meaning return `T` if the JSYS skips, `NIL` if not.

Because of these differences, caution must be exercised in turning `JSYS` calls into `JS` calls.

The symbolic JSYS name is looked up on the list `JSYSES`, an association-list with elements of the form `(`*JSYSNAME JSYSNUMBER #SKIPS*`)`. If no entry is found, then the file `STENEX.MAC` (or `SYS:MONSYMS.MAC` for Tops-20) is scanned.

Examples: `(JS BIN (OPNJFN` *FILE*`) NIL NIL 2)` returns the value of `AC2` after doing a `BIN` from the JFN of *FILE*. `(JS BOUT (OPNFJN` *FILE*`) 3)` sends a control-C to *FILE*. The value of this `JS` call is garbage.

`(XWD` N_1 N_2`)` [Function]

Returns `(LOGOR (LLSH` N_1 `18) (LOGAND` N_2 `777777Q))`, i.e. the word with N_1 in the left half and N_2 in the right.

`(BIT` *BIT# WORD*`)` [NoSpread Function]

If *WORD* is not specified, *BIT* simply returns a number with bit *BIT#* set to 1 and all other bits 0. If *WORD* is given, then `BIT` is a predicate that returns `T` if *BIT#* is set in *WORD*. Bits are numbered from left to right.

Examples: `(BIT 32)` is 8 (=10Q), `(BIT 32 8)` is `T`.

(JSYSERROR *ERRORN*) [NLambda Function]

Returns the TENEX/TOPS-20 error number for *ERRORN*. For example, (JSYSERROR GJFX23) is 600103Q. JSYSERROR compiles open as a constant.

This package also defines the following ASSEMBLE macros:

(JS *JSYSNAME*) Can be used in ASSEMBLE statements instead of (JSYS *JSYSNUMBER*).

(CV *EXPR*) Expands to (CQ (VAG (FIX *EXPR*))), which unboxes *EXPR* to AC1.

(CV2 *EXPR*) Expands to (CQ2 (VAG (FIX *EXPR*))), which unboxes *EXPR* to AC2, saving AC1.

23.13 NOBOX

Note: Nobox is a LispUsers package that is contained on the file NOBOX.COM. *It only works with Interlisp-10.*

This package contains facilities for subverting the normal manner of dynamically allocating and collecting CONS cells, large integer boxes, and floating boxes in Interlisp-10 by using static, compile-time allocation. Storage allocation is controlled by allocating the memory for temporary results (e.g. a list that will be thrown away or a floating number that will not exist outside a local computational context) at compile-time or load-time. This "static" storage will be reused whenever the given line of code is re-executed. Because functions which use these facilities may exhibit bizarre behaviour if they are called recursively or if values escape outside of them, these facilities must be used with extreme caution, and should be reserved for those cases where the normal method of storage allocation and garbage collection is not workable or practical. Note: compiled functions need no run-time support for these facilities, i.e. NOBOX does not have to be loaded to execute compiled code.

23.13.1 CONS Cells

The function CBOX may be used to avoid allocation of CONS cells. When run interpreted, CBOX is exactly equivalent to the function CONS. Compiled, CBOX operates like CONS, except that the CONS cell returned is constructed (once) at compile or load time. New values for CAR and CDR are smashed into the cell at each execution.

The function LBOX performs an analagous role for LIST. When run interpreted, LBOX is exactly equivalent to LIST. Compiled, the corresponding CONS cells are allocated at compile or load time. For example, (LBOX A B C) will cause a 3-element static list to be included with a compiled function's literals. Each time the corresponding compiled code is executed, those three cells will be returned containing the current values of the variables A, B, and C.

LBOX allocates as many cells as there are arguments in the corresponding form, i.e. the number of scratch cells is determined at compile time. The iterative statement operator SCRATCHCOLLECT enables avoiding CONSes when the length of a list is not known at compile-time. SCRATCHCOLLECT is used in iterative statements exactly as COLLECT. Each time it is executed, it reuses the cells that it returned on previous executions, which it remembers as an internal scratch list. The length of this scratch list is always the length of the longest value that was ever returned; new cells are allocated whenever the scratch list runs out, and they are permanently remembered.

The SCRATCHCOLLECT i.s.opr and the function SCRATCHLIST (page 14.2) have similar applications. With SCRATCHLIST, the user makes explicit the origin of the list getting smashed, while with the SCRATCHCOLLECT i.s.opr, the scratch list is hidden (and there is a different scratch-list for each occurence of the i.s.opr).

23.13.2 Number Boxes

The functions IBOX, FBOX, and NBOX, and the record declarations IBOX and FBOX are provided to improve the efficiency of arithmetic computations. They permit information to be given to the Interlisp-10 compiler that will inhibit the allocation (and subsequent collection) of number boxes needed for holding temporary results of numeric computations.[39] In addition, access time to variable-values that are known to be large integers or floating point numbers is improved.

The records IBOX and FBOX essentially describe the structure of large integer and floating point boxes respectively. IBOX consists of a single field, called I, which corresponds to the actual contents of the large integer box. FBOX consists of a single field, called F, which corresponds to the contents of the floating point box. For example, the user can create a large integer box containing a given value and assign it to X by saying (SETQ X (create IBOX I ← *FORM*)). Even if the value of *FORM* is a *small* integer, the result will be stored in a new, large number box. This seeming inefficiency is important because if *some* values of *FORM* might be large, making *all* values large means that the compiler can be told how to treat all references to X without generating run-time tests to discover how to do the unboxing. Thus, wherever the value of X is to be referenced, the user simply writes (fetch I of X). In compiling this expression, the compiler generates a single MOVE instruction without any type-testing whatsoever. The user can reuse that number box by saying (replace I of X with (FOO)), which is equivalent to, but much more efficent than, (SETN X (FOO)). In other words, once it is known that X is bound to a large integer, (replace I of ···) can be used in all number-contexts to inform the compiler of that fact.

The facilities described so far do nothing to suppress the creation of unnecessary boxes; indeed, the (create IBOX --) will produces boxes for small numbers that would not be allocated otherwise. The functions (not records) IBOX, FBOX, and NBOX are used to suppress unnecessary boxing of temporaries. Effectively, they cause "constant" or "static" boxes of the appropriate type to be allocated and stored in a function's literals when a function is compiled or loaded. Those boxes can be used (and reused) to hold temporary results.

IBOX and FBOX can be called with 0 or 1 arguments. If no arguments are specified (as opposed to a single argument whose value is NIL), then the value of the function is a large-integer or floating number box which is allocated statically. For example, these might be used to construct an initial binding for a variable into which temporary values will be stored using the I or F assignments. For example:

```
(PROG ((X (IBOX))) (replace I of X with (FOO)) ···)
```

[39] In the latter respect, these duplicate some of what SETN (page 22.5) does, except that they are more convenient to use and are executed with less run-time checking (i.e. SETN will never smash random memory locations).

If an argument is specified for IBOX or FBOX, then a static box of the appropriate type will be allocated at compile- or load-time, and the value of the argument will be stored in that box whenever the IBOX statement is executed. For example, suppose the user wanted to set a file pointer to 1 past a given byte position. The expression

```
(SETFILEPTR FILE (ADD1 POS))
```

would generate a new number box on each execution for which POS happened to be a large number. That box would be passed into SETFILEPTR and then returned as its value. Since the value is not saved, the box would be thrown away, to be collected later. The expression

```
(SETFILEPTR FILE (IBOX (ADD1 POS)))
```

would store the desired position in a constant box, and no allocations would take place.

As another example, consider a complicated integer expression whose value must be saved in a variable to be used a little further down in a program:

```
(SETQ X (IPLUS 2000 (ITIMES FOO (IQUOTIENT FUM 5))))
•
•
•
(SETQ Z (IPLUS X (GETFILEPTR FILE)))
```

The Interlisp-10 compiler is smart enough to suppress the boxing inside the (IPLUS 2000 &) expression, but it will generate a box when it comes to do the SETQ. This box can be suppressed by writing

```
(SETQ X (IBOX (IPLUS 2000 (ITIMES FOO (IQUOTIENT FUM 5)))))
```

Furthermore, since it is known that X is bound to a large integer, the Z assignment can be speeded up by writing

```
(SETQ Z (IPLUS X:I (GETFILEPTR FILE)))
```

The function FBOX behaves the same as IBOX, except that it uses constant floating boxes. Note that if the argument of IBOX is FLOATP, then it will be FIXed; if the argument of FBOX is FIXP, it will be FLOATed.

The function NBOX is a generic function for copying unknown values into constant number boxes. It allocates two constant boxes, one integer and one floating, and stores the value of its argument in the one compatible with the value's type. NBOX is useful if the argument value is a constant number box (but one of unknown type) that needs to be copied (see caution (2) below).

23.13.3 Cautions

There are some dangers in using these facilities. The user of this package should be particularly aware of the following:

(1) The F and I fields aim at efficiency more than validity. This means that they *do not check the type* of

the pointer that they smash into. For example, if X is bound to NIL, the expression (replace I of X with Z will clobber CAR and CDR of NIL! The user must be very careful that the arguments given for replacing do indeed point to cells that unboxed numbers can be smashed into. Note: the DECL package (page 23.18) can be used to generate the replaces, IBOXes, FBOXes automatically in a safe and efficient way.

(2) CBOX, LBOX, SCRATCHCOLLECT, IBOX, and FBOX all allocate constant boxes, and those boxes will be reused (i.e. smashed with new values) every time the code containing that function call is executed. If that box is saved in a variable or data-structure (e.g. by a SETQ) as a way of preserving the value it contains, and then the code is re-executed, the value that was saved will be smashed. Thus, the user must beware of using constant boxes to save information in loops or recursions that can get back to the same statement. In these situations, the values must be copied into other cells, perhaps a constant associated with some other line of code, or into cells allocated in the ordinary way. The user must also be careful about returning a constant box as the value of a function, since the caller might unknowingly save the value and re-invoke the box-returner.

(3) Because the constant boxes are allocated only in compiled code, *these functions will work quite differently compiled and interpreted.* Side effects which occur because of inadvertent smashing of shared structures will only occur when running compiled definitions and will not be detectable when running interpreted.

23.14 DATEFORMAT

Note: Dateformat is a LispUsers package that is contained on the file DATEFORMAT.COM. *It only works in Interlisp-10.*

Dateformat is a small file (one function) which provides assistance for constructing format bits for the ODTIM JSYS (output date/time) as required by DATE and GDATE (page 14.9).

(DATEFORMAT *KEY*$_1$ ··· *KEY*$_N$) [NLambda NoSpread Function]

KEY$_1$ ··· *KEY*$_N$ are a set of keywords (unevaluated). DATEFORMAT returns a number suitable as a parameter to DATE and GDATE. The variable DATEFORMAT.DEFAULT is the number used as the initial value to work with. Therefore, to switch any of the defaults, set the variable DATEFORMAT.DEFAULT to be the value of a call to DATEFORMAT with the appropriate keys.

The keywords are given below (usually in pairs) and can be thought of as switches (i.e. turn on or off a particular format feature). If no keyword is given for a particular pair, the default is used.

The variable DATEFORMAT.KEYS is a list of the keywords used for spelling correction.

DATE (default)
NO.DATE Do/don't include the date information.

NAME.OF.MONTH (default)
NUMBER.OF.MONTH
Show the month as a name (NAME.OF.MONTH) or a number (NUMBER.OF.MONTH).

MONTH.LONG

MONTH.SHORT (default)
If the name of the month was requested, spell it out (MONTH.LONG) or abbreviate it (MONTH.SHORT).

YEAR.LONG
YEAR.SHORT (default)
Print four digit year, e.g. 1978 (YEAR.LONG) or two digit year, e.g. 78 (YEAR.SHORT).

DAY.OF.WEEK
NO.DAY.OF.WEEK (default)
Do/Don't include the day of the week in the date information.

DAY.LONG
DAY.SHORT (default)
If the day of the week was included, spell it out (DAY.LONG) or abbreviate it (DAY.SHORT).

DASHES (default)
SLASHES
SPACES
Separate the <day>, <month>, and <year> fields with dashes/slashes/spaces.

USA.FORMAT
EUROPE.FORMAT (default)
Print the date in the order <month> <day> <year> (USA.FORMAT) or in the order <day> <month> <year> (EUROPE.FORMAT).

LEADING.SPACES (default)
NO.LEADING.SPACES
If LEADING.SPACES is specified, the <day> field will always be two characters long. If NO.LEADING.SPACES, the <day> field can be one character for dates earlier than the 10th.

TIME (default)
NO.TIME
Do/Don't include the time information.

TIME.ZONE
NO.TIME.ZONE (default)
Do/Don't include the time zone in the time specification.

SECONDS (default)
NO.SECONDS
Do/Don't include the seconds.

CIVILIAN.TIME
MILITARY.TIME (default)
Use 12 hour time with AM or PM (CIVILIAN.TIME) or 24 hour time

(MILITARY.TIME).

23.15 EXEC

Note: The Exec package is a LispUsers package that is contained on the file EXEC.COM. *The Exec package uses the passwords package (see page 23.62). Loading* EXEC.COM *will load* PASSWORDS.COM *if it has not already been loaded. Note: some of the facilities described below will work correctly only on TENEX systems, others only on TOPS-20. The system will inform the user when he attempts to use a facility not supported by his particular operating system.*

This package defines a set of programmer's assistant commands which resemble features of the Tenex EXEC. It also defines functions that provide certain EXEC capabilities for Interlisp programs, e.g. changing the connected directory, detaching the job, etc.

23.15.1 Exec Commands

DA [Exec Command]

Prints out the current time and date.

LD [Exec Command]
SY [Exec Command]
WHE [Exec Command]

Prints SYSTAT information, just like the LD subsystem. Jobs are sorted in inverse order of CPU utilization.

LD *USERNAME* [Exec Command]

Prints information for the specified user only.

LD ALL [Exec Command]

Like LD, but includes system jobs.

DET [Exec Command]

Detaches the current job.

QU [Exec Command]

Does a (LOGOUT). Does not go on history list.

LINK *USER* [Exec Command]
TALK *USER* [Exec Command]

Mimics the exec link command. If *USER* has multiple jobs logged in, asks which tty to link to.

BR [Exec Command]

Breaks links.

CONN *DIR PWD* [Exec Command]

Connects to the directory *DIR*. If the password *PWD* is not given and is required, CONN will prompt. *DIR* can be abbreviated; if omitted, it defaults to the user's login directory. If *PWD* is given in command line, it is removed from the history list so that ?? will not print it out. Password prompting is handled by GETPASSWORD from the passwords package (page 23.62).

NDIR *FILEGROUP* [Exec Command]

Prints the files in *FILEGROUP* in a multi-column format.

NDIR *FILEGROUP* [Exec Command]

Deletes specified files. Uses DIRECTORY (page 14.6). Note that if <esc> is specified, *all* files that match will be deleted. This command is undoable.

UND *FILEGROUP* [Exec Command]

Undeletes the specified files (undoably).

DELVER *FILEGROUP* [Exec Command]

Deletes all but 1 version of the filegroup specified. Uses DIRECTORY (page 14.6), so *FILEGROUP* may utilize any of the options allowed for directory filegroup specifications.

EXP *DIR* [Exec Command]

Expunges directory *DIR*. If the user does not have access to *DIR*, a message is printed.

TY *FILE OUTFILE BYTESIZE* [Exec Command]
SEE *FILE OUTFILE BYTESIZE* [Exec Command]

Copies *FILE* to *OUTFILE*, or to T if OUTFILE is not given. Assumes that the bytes of *FILE* are *BYTESIZE* bits wide (*BYTESIZE*=NIL defaults to 7). Suppresses blank lines and control character sequences used to indicate font changes.

DSK *DIR DAYS* [Exec Command]

Prints out disk allocation and usage for the directory *DIR* using DSKSTAT. Also prints total size of files untouched in days *DAYS* (90 if *DAYS* not specified).

FI [Exec Command]

Like the EXEC FILESTAT command, prints out status of all currently assigned JFNS for the current job.

FI *JFN* [Exec Command]

Prints information for *JFN* only.

23.15.2 EXEC Functions

(JOB#) [Function]

Returns the job number for the logged in job.

(TTY#) [Function]

Returns the teletype-number of the current job.

(DETACH) [Function]

Detaches the current job.

(DETACHEDP) [Function]

Returns T if the current program is running detached.

(LINKTOTTY *TTY#*) [Function]

Generates a two-way link between the controlling terminal of the user's job and *TTY#*. Returns T if the link was successful, otherwise prints an error message and returns NIL.

(LINKTOUSER *USER*) [Function]

Links the controlling terminal to a terminal associated with *USER*. Generates an error if the user is not logged in or not attached. If *USER* has more than one attached job, then a systat of his jobs is printed, and the user is asked to provide the proper tty number for the job. Returns T if successful.

(BREAKLINKS) [Function]

Breaks all links to the user's controlling terminal.

(CNDIR *DIR PASSWORD*) [Function]

Implements the CONN command.

(/DELFILE *FILE*) [Function]

Undoable version of DELFILE.

(/UNDELFILE *FILE*) [Function]

Undeletes a single file (undoably).

(EXPUNGE *DIR*) [Function]

Expunges directory *DIR*. On TENEX, *DIR* is ignored. and the connected directory is expunged. On TOPS20, if the user does not have access to *DIR*, a message is printed.

(COPYALLBYTES *FROMFILE TOFILE BYTESIZE*) [Function]

Implements the SEE command.

(DSKSTAT *DIR PRINTIFOVER PRINTSYS PRINTDEL PRINTOLD*) [Function]

Prints disk usage statistics for directory *DIR* (either a name or number).

If *PRINTIFOVER* is NIL, this means always print. If *PRINTIFOVER* is T, this means only print if *DIR* is over allocation. If *PRINTIFOVER* is a number, this means only print if *DIR* has more than that many pages in use.

If *PRINTSYS* is T, this means print system disk statistics too.

If *PRINTDEL* is T, this means print total size of deleted files for *DIR* (this is slow).

If *PRINTOLD* is T or a number, this means print total size of files untouched in 90 (or *PRINTOLD*) days.

(MEMSTAT *PG1 PGN FORK*) [Function]

Prints the status of the memory pages *PG1* (0 if *PG1*=NIL) to *PGN* (the last page of memory if NIL) in fork *FORK*. *FORK* is either NIL, meaning the current fork, or a fork handle.

23.16 PASSWORDS

Note: Passwords is a LispUsers package that is contained on the file PASSWORDS.COM. *It only works with Interlisp-10.*

(GETPASSWORD *DIRECTORYNAME*) [Function]

Prompts the user for the password for the given directory. The user's response is not echoed. GETPASSWORD remembers the password so that it need not ask again; however, saved information is cleared before SYSOUT, so that the SYSOUT contains no passwords.

23.17 TELNET

Note: Telnet is a LispUsers package that is contained on the file TELNET.COM. *It only works with Interlisp-10. Since the telnet package uses the net package, loading* TELNET.COM *will also load* NET.COM *unless it has already been loaded.*

This package makes it possible to interact with connections created via the net package (page 23.64) without leaving Interlisp. In addition, all typeout is included in the DRIBBLE file. It permits connections to ARPANET hosts (a la TELNET).

(TELNET *CONNECTION TYPE SKT* —) [Function]

CONNECTION may be an instance of a CONNECTION record (as created by MAKENEWCONNECTION, page 23.64). Alternatively, if *CONNECTION* is a litatom, TELNET uses (MAKENEWCONNECTION *CONNECTION TYPE SKT*) for the connection. In any case, TELNET returns the connection as an instance of the CONNECTION record, so that it is possible to TELNET back.

23.18 FTP

Note: Ftp is a LispUsers package that is contained on the file FTP.COM. *It only works with Interlisp-10. Since the Ftp package uses the net and passwords packages, loading* FTP.COM *will also load* NET.COM *and* PASSWORDS.COM *if they are not already loaded.*

The ftp package makes it possible to deal with files at other hosts on the Arpa network almost as if they were files on the user's local machine, i.e. the files can be opened via INFILE, OUTFILE, OPENFILE, read from and printed to by the ordinary reading and printing functions, and closed in the standard way.

Files on remote hosts are designated by including the host name between curly brackets, {}, at the front of the ordinary file name. Since curly brackets are illegal characters in regular file names, a BAD

FILE NAME error is generated. This error is intercepted by an entry on ERRORTYPELST (see page 9.16) which then establishes the appropriate network connections.[40] For example, (INFILE '{BBN-D}<LEWIS>INIT.LISP) will open the file <LEWIS>INIT.LISP on the host BBN-D and make it be the primary input file. The user could then say (READ) to obtain the first expression on that file. The ftp package extends the functions PACKFILENAME, UNPACKFILENAME, and FILENAMEFIELD so that they will associate the curly bracket syntax with the new file field HOST. Thus, (PACKFILENAME 'HOST 'BBND 'NAME 'INIT) will return {BBND}INIT.

Remote files have certain properties that limit how they may be used:

(1) RANDACCESSP is NIL for such files, and SETFILEPTR may not be applied to them. This means, for example, that functions and variables may not be loaded from such files via LOADFNS.

(2) The open bytesize of a remote file may not be changed (e.g. by SETFILEINFO). This means that Interlisp-10 compiled files may not be loaded from remote hosts.

(3) The remote host may close the connection spontaneously (e.g. because of a timeout if the file is not referenced for some length of time, or because of a crash). If this happens, the next attempt at reading or writing on the file will generate FILE DATA ERROR. Note: it is unwise to keep a remote file open for long periods of time.[41]

When the connection for the remote file is first established, a password for the remote machine/directory may be required. The user will be asked to supply one via the passwords package (page 23.62). Alternatively, if the host name has on its property list the property LOGIN with value of the form (*NAME PASSWORD ACCOUNT*), then the indicated *NAME*, *PASSWORD*, and *ACCOUNT* will be used to log the user into the remote host.[42]

(FTP *HOST FILE ACCESS USER PASSWORD ACCOUNT BYTESIZE*) [Function]

Opens a network connection to the ftp server at *HOST*. If *ACCESS*=INPUT or OUTPUT, FTP works like OPENFILE: value is a literal atom of the form {*HOST*}*FILE* which can then be used as a file name by all Interlisp input and output functions, e.g. READ, PRINT, COPYBYTES, etc.[43] For example, (FTP 'SU-AI 'YUMYUM%[P,DOC%] 'INPUT) will allow the Stanford Restraurant Guide to be read. Note that *FILE* must satisfy the file name conventions of the remote host.

[40] Note: it is fairly expensive to open a network connection as compared with the time to open a local file, e.g. an order of magnitude slower.

[41] For input files, these limitations may be skirted conveniently in the following way: if a colon appears between the last character of the host name and the right curly bracket (e.g. {BBND:}<LEWIS>INIT.LISP), then the remote file will be copied to a temporary *local* file when it is opened, and all subsequent references will be to that local file.

[42] If the value is of the form (*NAME* NIL *ACCOUNT*), then (GETPASSWORD *NAME*) will be used for the password. If the *ACCOUNT* field is NIL, no account will be supllied to the remote host. If no LOGIN property is supplied, ANONYMOUS will be used as the user name.

[43] In reality, this "file" is a network connection to the host's ftp server. This "file" has a WHENCLOSE attribute (page 6.11) associated with it so that when Interlisp closes the file, the correct terminating sequence will be performed.

If *ACCESS*=DIRECTORY, then FTP will print on the terminal the names of all files which match *FILE*, e.g. (FTP 'PARC-MAXC2 '<NETLISP>*.SAV 'DIRECTORY).

USER, *PASSWORD*, and *ACCOUNT* are used for logging in to the remote host. If not supplied, the values are obtained from the LOGIN property (if any) as described above. *BYTESIZE* is the byte size in which to open the connection. Byte sizes of 7, 8, 16, 32 and 36 are supported. *BYTESIZE*=NIL defaults to 7.

23.19 NET

Note: Net is a LispUsers package that is contained on the file NET.COM. *It only works with Interlisp-10.*

This package contains functions for establishing ARPANET connections from an Interlisp-10 job. A connection is described by and is an instance of the record CONNECTION. The only fields of interest to the user in this record are IN and OUT, which are guaranteed to be CAR and CADR, respectively. IN is a file name which can be read from, OUT a file name which can be printed to.

(MAKENEWCONNECTION *HOST TYPE SKT SCRATCHCONN WAITFLG*) [Function]

Makes a connection to *HOST*. For *TYPE*=ARPA, *HOST* is the name of the host to which the connection is to be made. For *SKT*=NIL (the normal case), the connection will be to the telnet server of *HOST*; connections to other servers can be made by supplying the appropriate value for *SKT*.

The value of MAKENEWCONNECTION is a CONNECTION. If *WAITFLG* is non-NIL, MAKENEWCONNECTION waits until its request for connection is acknowledged. Otherwise, CHECKCONNECTION must be called on the result before it is used (this allows additional processing to be done while waiting for the remote host to respond).

If *SCRATCHCONN* is non-NIL, it is a scratch connection which is reused.

For example, (MAKENEWCONNECTION 'BBND) makes an ARPA connection to BBND, (MAKENEWCONNECTION 'SU-AI 'ARPA 'FINGER) makes a connection to the Stanford WHEREIS service.

(CLOSECONNECTION *CONNECTION*) [Function]

Closes the given *CONNECTION* and replaces the IN and OUT fields with NIL.

(CHECKCONNECTION *CONNECTION*) [Function]

Checks to make sure that the given connection is still open (e.g. it hasn't been closed remotely). If the connection is valid, *CONNECTION* is returned. If the connection is in an in-between state, i.e. in the process of being opened or closed, CHECKCONNECTION waits to see what happens before returning. Otherwise the connection is cleaned up (as if a CLOSECONNECTION were performed) and CHECKCONNECTION returns NIL.

(NETSERVER *ARPA# WAITFLG*) [Function]

Initiates a "server" connection. This is a connection which will talk to a "user" connection. If *WAITFLG* is non-NIL, waits for a user to connect; if *WAITFLG*=NIL, returns immediately (and CHECKCONNECTION must be called on the connection before the connection is actually used). *ARPA#* defaults to 0.

(NETUSER *HOST USER ARPA# WAITFLG*) [Function]

Initiates the other half of an Arpa connection. *ARPA#* defaults to 0 and must be the same as the argument given the corresponding call to NETSERVER. *USER* must be the USERNUMBER (directory number) under which the server job is logged in.

For example, to establish an ARPANET connection between two Interlisp jobs (which can then be written to and read from like files), do (SETQ CONN (NETSERVER)) in one job and (SETQ CONN (NETUSER *HOST USER*)) in the other job, where *HOST* is the machine on which the first job is running and *USER* is the directory number under which the first job is logged in (obtainable through the function USERNUMBER). Then, perform (CHECKCONNECTION CONN) in each job; when these return, the connection is ready to be used.

(FORCEOUT *CONNECTION/FILE*) [Function]

Normally, characters sent to the "OUT" of a connection are buffered locally. The function FORCEOUT can be used to force partially filled packets of bytes to be sent across the connection. The argument to FORCEOUT can either be the CONNECTION record or the OUT filename.

Net

INDEX

XEROX®

Headquarters

Xerox Special Information Systems
250 North Halstead Street
Pasadena, California 91107-7018
(213) 351-2351

Washington, D.C.

Xerox Special Information Systems
1616 North Ft. Myer Drive, 16th Floor
Arlington, Virginia 22209
(703) 841-0770

Northeast Region

Xerox Special Information Systems
950 Franklin Avenue
Garden City, New York 11530
(516) 294-6493

Northwest Region

Xerox Special Information Systems
1169 Lea Drive
Novato, California 94947
(415) 577-0435

Printed in U.S.A.