Macintosh®

Macintosh Programmer's Workshop 2.0 Pascal

20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Contents

Tables and Figures

Preface

WELCOME TO THE *MACINTOSH PROGRAMMER'S WORKSHOP 3.0 PASCAL REFERENCE.* This manual contains complete reference material on the Macintosh Programmer's Workshop implementation of the Pascal language (called *MPW Pascal*), as well as material on the Pascal Compiler and the libraries of predeclared procedures and functions that are part of the MPW Pascal system. ■

Contents

About APDA

APDA™ is an excellent source of technical information for anyone interested in developing Apple-compatible products. Membership in the association allows you to purchase Apple technical documentation, programming tools, and utilities. For information on membership fees, available products, and prices, please contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33–G
Cupertino, CA 95014–6299

1-800-282-APDA, or 1-800-282-2732
Fax: 408-562-3971
Telex: 171-576
AppleLink: DEV.CHANNELS

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 51–W
Cupertino, CA 95014–6299

User groups

Ask your authorized Apple dealer for the name of the Macintosh user group nearest you, or call 1-800-538-9696. for information about starting your own user group, contact either:

The Boston Computer Society
One Center Plaza
Boston, MA 02108
USA
(617) 367-8080

or

Berkeley Macintosh User's Group
1442-A Walnut Street #62
Berkeley, CA 94709
USA
(415) 849-9114

About this manual

This manual provides information about the MPW Pascal language and the use of the MPW 3.0 Pascal programming system. Here is a brief description of each chapter and appendix:

- Chapter 1, "About MPW Pascal," contains general information about the MPW 3.0 Pascal language and Compiler and tells about the files you use to build an application for an Apple® Macintosh™ computer.
- Chapter 2, "Symbols," describes the fundamental components of the Pascal language.
- Chapter 3, "Blocks and Scope," explains the block-structured nature of MPW Pascal and discusses its scope rules.
- Chapter 4, "Data Types," gives an overview of MPW Pascal's predefined data types and type constructors.
- Chapter 5, "Constants and Variables," describes the forms that variables can take within a Pascal program.
- Chapter 6, "Expressions," details the rules governing the structure of Pascal expressions and includes descriptions of the Pascal operators.
- Chapter 7, "Statements," defines and gives examples of each of the Pascal statement types.
- Chapter 8, "Procedures and Functions," tells how to declare procedures and functions and defines the use of parameters.
- Chapter 9, "Programs and Units," discusses the overall structure of Pascal programs and describes the use of units in writing large programs.
- Chapter 10, "Files and I/O," explains the use of files and the routines that perform input and output tasks in a Pascal program.
- Chapter 11, "Predefined Routines," provides information on the routines that are built into the MPW Pascal Compiler and the non-I/O routines that are included in PasLib.
- Chapter 12, "Object-Oriented Programming," describes the facilities provided in MPW Pascal for creating and manipulating objects.
- Chapter 13, "Compiler Options and Directives," contains information on Compiler options and directives.
- Appendix A, "MPW 3.0 Pascal and Other Pascals," explains how this version of Pascal relates to the ANSI Standard and other Apple versions of Pascal.
- Appendix B, "Special Scope Rules," covers MPW Pascal scope rules that are applicable in special situations.
- Appendix C, "Reserved Words and the Character Set," contains quick reference information on these topics.

- Appendix D, "Syntax Summary," lists all the syntax diagrams used in this book.
- Appendix E, "MPW 3.0 Pascal Files," is a complete list of the files that constitute the MPW Pascal system.
- Appendix F, "Pascal and C Calling Conventions," explains how the Compiler passes parameters and tells how to declare procedures using the C directive.
- Appendix G, "The SANE Library," describes the routines in the Pascal library that implement the Standard Apple Numerics Environment (SANE) and provides special information about the use of SANE and the 68881 floating-point coprocessor.
- Appendix H, "The PasMat Utility," tells how to use the Pascal utility program that converts your source text into standard format.
- Appendix I, "The PasRef Utility," tells how to use the Pascal utility program that generates a cross-referenced list of the identifiers in your program.
- Appendix J, "The ProcNames Utility," tells how to use the Pascal utility program that displays Pascal procedure and function names.
- Appendix K, "Advanced Topics for 68020 Programmers," gives special information for those programmers using the 68020 central processing unit.

Aids to understanding

Look for these visual cues throughout the manual:

▲ **Warning** Warnings like this indicate potential problems. ▲

△ **Important** Text set off in this manner presents important information. △

◆ *Note:* Text set off in this manner presents notes, reminders, and hints.

Computer words and phrases appear in boldface type when they are introduced. The term is defined in the Glossary.

Other reference materials

The following books contain important reference material that you'll need when writing programs in MPW Pascal:

- Apple Computer, Inc., *Apple Numerics Manual*, Addison-Wesley, 1986. A description of the Standard Apple Numeric Environment and how it is invoked in the Macintosh.
- Apple Computer, Inc., *Inside Macintosh* (Volumes I–III), Addison-Wesley, 1985. The complete story of the architecture and operation of the 128K and 512K Macintosh, including details of its ROM routines.
- Apple Computer, Inc., *Inside Macintosh* (Volume IV), Addison-Wesley, 1986. Additional and updated material covering the Macintosh and Macintosh Plus.
- Apple Computer, Inc., *Inside Macintosh* (Volume V), APDA, 1987. Additional and updated material covering the Macintosh II and Macintosh SE.
- Apple Computer, Inc., *Macintosh Programmer's Workshop 3.0 Reference*, APDA, 1988. A full description of how to use the MPW program preparation tools, including the Pascal Compiler.

In addition, you may find the following books helpful as a supplement to this manual:

- Henry Ledgard, *The American Pascal Standard, with Annotations*, Springer-Verlag, 1984. An annotated guide to ANS Pascal, as defined by the American National Standards Institute.
- Apple Computer, Inc., *MacApp 2.0 Programmer's Reference*, APDA, 1987. How to use MacApp™ with Object Pascal. For a brief description of MacApp, see Chapter 12.
- Apple Computer, Inc., *Macintosh Programmer's Workshop 3.0 Assembler Reference*, APDA, 1988. How to write assembly-language programs that you can link with MPW Pascal.
- Apple Computer, Inc., *Macintosh Programmer's Workshop 3.0 C Reference*, APDA, 1988. How to write C programs that you can link with MPW Pascal.
- Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Co., 1986. A comprehensive introduction to Object Pascal and the theory behind MacApp.
- Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, 3rd edition, Springer-Verlag, 1985. Revised by Andrew B. Mickel and James F. Miner. The original, and in many ways best, definition of Pascal.

You may want to find further information about the MC68020 and the MC68881 in these volumes:

- Motorola, *MC68020 32-Bit Microprocessor User's Manual*, 2nd edition, Prentice-Hall, 1985. The latest complete information for engineers, software architects, and computer designers working on hardware and software systems using the MC68020.
- Motorola, *MC68881 Floating-Point Coprocessor User's Manual*, 1st edition, Motorola, 1985. The latest complete information for engineers, software architects, and computer designers to aid in the implementation of hardware and software systems using the MC68881.

Notation

This manual uses typographic conventions to distinguish between different types of words and symbols. Four fonts are used:

- Ordinary English is printed in plain Roman letters, the kind you are reading now.
- Special technical terms are printed in **boldface** when they are first defined. After that, they are treated as ordinary English. Such terms are also defined in the Glossary at the end of this manual.
- Elements of the Pascal language (or any other computer language) are printed in `computer voice`. This helps you avoid confusing them with ordinary English words.
- Artificial terms, which have meaning only in this book, are printed in *italics*. Such terms are sometimes called *metasymbols;* they are used primarily to indicate parts of syntax diagrams that you replace with actual Pascal symbols.

Within the Pascal language, using the computer voice font, the following capitalization conventions are used:

- Reserved words are printed in `ALL CAPITALS`.
- The names of predefined procedures and functions (that is, those that are part of the MPW Pascal language) are printed in `Initial Capitals`.
- The names of data types and constants are printed in `lowercase`.

Here is an example of how these fonts work together:

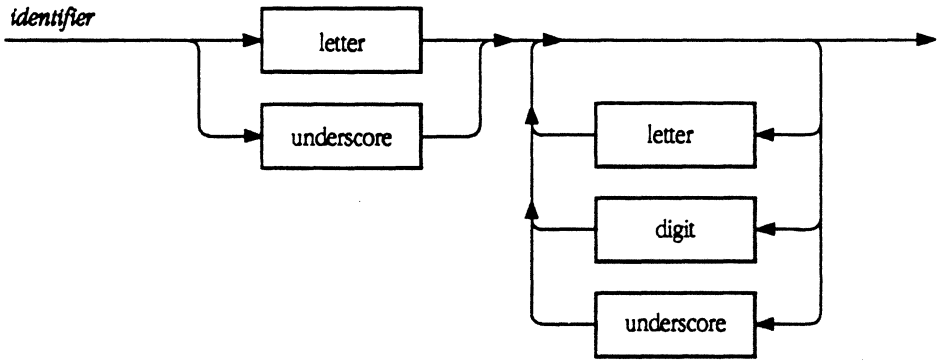
"The value of each `Write` parameter, p_n, is given by an **output expression,** which may be of type `char`, `integer`, `real`, `STRING`, `PACKED ARRAY OF char`, or `boolean`."

Syntax diagrams

Throughout this manual, the syntax of MPW Pascal is illustrated with syntax diagrams. These diagrams show you the rules that govern the way the elements of the language are used. Figure P-1 is an example of a syntax diagram.

Within the syntax diagrams, words enclosed in rounded bubbles are reserved words or other Pascal symbols. Words enclosed in boxes with square corners are higher-level constructs, many of which have their own syntax diagrams.

- **Figure P-1** Example of syntax diagram (identifier syntax)



This diagram shows that an identifier begins with a letter or an underscore, and that this letter may be followed by a letter, a digit, an underscore, or nothing. From there, you can loop back to add another letter, digit, or underscore, or nothing at all.

The notation used to describe the syntax of predefined procedures and functions is different. Here's an example of the format:

`Write` (*f*, p_1 [, p_2, ..., p_n])

This represents the actual syntax of the predefined procedure `Write`. Notice the following details:

- The terms *f*, p_1, p_2, and p_n stand for actual parameters. The types and interpretations of the parameters are given in the discussion of each procedure or function.
- The notation "p_1, p_2, ..., p_n" means that any number of actual parameters can appear here, separated by commas.
- Square brackets, [ ], indicate parts of the syntax that can be omitted.

Hence the example shows that you must pass to the procedure `Write` parameters that correspond to *f* and p_1. Additional parameters are optional.

Ellipses

A sequence of three dots (...) in a syntax diagram indicates repetition of the preceding material.

A sequence of two dots (..) indicates a scalar range. For example, 0..127 means "0 through 127."

A sequence of three hyphens (---) in a sample source text listing indicates lines not specified in the sample.

Chapter 1 About MPW Pascal

MPW 3.0 PASCAL IS AN IMPLEMENTATION of the Pascal language that is part of the Macintosh Programmer's Workshop 3.0. It consists of several disk files:

- the MPW 3.0 Pascal compiler
- three special tools, PasMat and PasRef, for formatting and cross-referencing Pascal programs (described in Appendixes H and I, respectively) and ProcNames, for producing lists of the procedures and functions in your Pascal programs or units (described in Appendix J)
- files of interface declarations that provide access to the Pascal, SANE, and Macintosh routines
- the Pascal and SANE libraries
- several sample programs, with instructions for building them (including the sample program, TestPerf.p for the performance tool PerformReport, which is located in the Tools folder on the MPW 3.0 disk)

A complete list of the MPW 3.0 Pascal files is included in Appendix E. ■

Contents

About MPW Pascal version 3.0

MPW 3.0 Pascal is a replacement version of MPW Pascal 2.0. If you're familiar with MPW Pascal 2.0, see Appendix A for a list of the differences between the two versions. Appendix A also contains a compliance statement about MPW Pascal's relationship to the American National Standards Institute's definition of Pascal (ANS Pascal).

Besides providing nearly all the capabilities of Pascal described in the ANS Pascal Standard, MPW 3.0 Pascal includes the following new features that expand the power and flexibility of Pascal programming:

- support for SADE, the symbolic debugger (described in Chapter 13)
- a replacement for the `$LOAD` directive (described in "Automatic Symbol Loading" in Chapter 9 and "The `-noload`, `-clean`, and `-rebuild` options" in Chapter 13)
- the use of character constants as valid string expressions
- extended and improved symbol support for MacsBug (described in Chapter 13)
- support for greater than 32K global data (described in Chapter 13)
- less strict requirements for forward class references
- new interface file organization

About SADE and MacsBug

The new Symbolic Apple Debugging Environment (SADE) is a symbolic debugger with an interactive graphic interface like that of the MPW Shell. You can monitor the execution of your program simultaneously at the processor level and the symbolic program source level. This first release of SADE includes

- source display and source breakpoints
- variable display according to type (including records)
- display of Macintosh system structures
- programmable, extensible command language

SADE is included with the MPW 3.0 program but documented separately in the *SADE Reference*. The familiar MacsBug application has been improved for MPW 3.0, and is also documented in a separate volume, *MacsBug Reference*.

MacsBug fully supports the MC68000 and MC68020 processors, as well as the MC68881 and MC68851 coprocessors. It is installed at startup, resides in RAM with your computer, and runs on all Macintosh computers, including the Macintosh SE and the Macintosh II. With MacsBug, you can examine memory, trace through a program, or set up break conditions and execute a program until these conditions occur. See the *SADE Reference* for instructions on using MacsBug and Appendix F of the *Macintosh Programmer's Workshop 3.0 Reference* for the object file format.

Object Pascal

MPW Pascal includes a set of extensions, collectively known as **Object Pascal**, that provide you with the ability to write object-oriented programs.

Object-oriented languages, such as Smalltalk-80 and Simula-67, let you structure your programs in ways that allow for greater control over the ways they process data. Object-oriented programming couples data and routines to produce powerful, easily maintainable code. It also gives you the ability to write programs using MacApp, Apple's "expandable" Macintosh application.

MacApp provides a skeleton Macintosh application. It supplies a framework that implements many of the features of the Macintosh interface, to which you add the unique features of your own application. See the *MacApp 2.0 Programmer's Reference* for more information.

The Object Pascal extensions are described at various places in this manual. If you're new to object-oriented programming, you may want to read an introductory book on the subject before you attempt to use Object Pascal. For suggestions, see "Other Reference Materials" in the Preface.

The philosophy behind object-oriented programming is summarized briefly in Chapter 12.

Link, the Linker tool described in the *Macintosh Programmer's Workshop 3.0 Reference*, now contains the optimizing code for Object Pascal. It is available as the `-opt` option, and it eliminates any need for the Optimize tool distributed with MacApp.

MPW Pascal provides strict error reporting for object errors. For details, see "Compiler Options" in Chapter 13.

About the Pascal interface files

The MPW 3.0 Pascal interface files contain declarations for the routines in the MPW 3.0 Pascal libraries and the MPW 3.0 libraries, as well as the User Interface Toolbox and Operating System routines that are built into the Macintosh ROM. The Macintosh ROM routines are described in detail in *Inside Macintosh*, Volumes 1 through 5. The interfaces to these routines are divided into files according to their "Manager," as described in *Inside Macintosh*.

Here is a list of the changes in the Pascal interface files since MPW Pascal 2.0:

- Toolbox and operating system interfaces have been divided into files according to Manager rather than being divided between `ToolIntf.p` and `OSIntf.p`. This parallels the organization of Inside Macintosh as well as the C include files.
- `MemTypes.p`, `OSIntf.p`, `ToolIntf.p`, `PackIntf.p`, `PickerIntf.p`, `SCSIIntf.p`, and `VideoIntf.p` are retained for compatibility; however, they have been modified to include the appropriate new interface files. It is often preferable to use the new interfaces directly. It is unlikely that you will need all of the new interfaces previously included in `OSIntf.p` and `ToolIntf.p`, so only use the new interfaces that your program depends upon.
- The new interface files will include the files that they depend upon, if necessary.
- `Sound.p` has been updated to include all the Macintosh sound routines previously included in `Sound.p` and `OSIntf.p`.
- `Printing.p` and `PrintTraps.p` perform essentially the same function; however, `Printing.p` checks to find out if it can use the apppropriate ROM routines and includes the necessary glue to work with 64K ROMs. `PrintTraps.p` generates more efficient code that calls the ROM directly.

See Appendix G for more about SANE and the MC68881.

Table 1-1 lists the new interface files.

■ **Table 1-1** New interface files used in MPW Pascal

Interface file	Contents
`Controls.p`	Control Manager interface
`Desk.p`	Desk Manager interface
`DeskBus.p`	Apple Desktop Bus Manager interface
`Devices.p`	Device Manager interface
`Dialogs.p`	Dialog Manager interface
`DisAsmLookup.p`	SADE and MacsBug symbols
`DiskInit.p`	Disk Initialization package interface
`Disks.p`	Disk Driver interface
`Errors.p`	Error file
`Events.p`	Event Manager interface
`Files.p`	File Manager interface
`Fonts.p`	Font Manager interface
`HyperXCmd.p`	HyperCard 'XCMD' interface
`Lists.p`	List Manager interface
`Memory.p`	Memory Manager interface
`Menus.p`	Menu Manager interface
`Notification.p`	Notification Manager interface
`OSEvents.p`	Operating System Event Manager interface
`OSUtils.p`	Operating System Utilities interface
`Packages.p`	Package Manager interface
`Palettes.p`	Palette Manager interface
`Picker.p`	Color Picker Manager interface
`Printing.p`	Printing interface
`Resources.p`	Resources Manager interface
`Retrace.p`	Vertical Retrace Manager interface
`Scrap.p`	Scrap Manager interface
`SCSI.p`	SCSI Manager interface
`SegLoad.p`	Segment Loader interface
`Serial.p`	Serial Driver interface
`Shutdown.p`	Shutdown Manager interface
`Slots.p`	Slot Manager interface
`Start.p`	Start Manager interface
`Strings.p`	String conversion routines

(Continued)

■ **Table 1-1** (Continued) New interface files used in MPW Pascal

Interface file	Contents
TextEdit.p	Text Edit interface
Timer.p	Timer Manager interface
ToolUtils.p	Toolbox Utilities interface
Types.p	Common types
Video.p	Video interface
Windows.p	Window Manager interface

Table 1-2 lists the old Pascal interfaces along with the new interfaces to use directly.

■ **Table 1-2** Interface files included for compatibility in MPW Pascal

Instead of this file	Use a subset of
MacPrint.p	Printing.p
MemTypes.p	Types.p
OSIntf.p	OSUtils.p, Events.p, Files.p, Devices.p, DeskBus.p, DiskInit.p, Disks.p, Errors.p, Memory.p, OSEvents.p, Retrace.p, Segload.p, Serial.p, Shutdown.p, Slots.p, Sound.p, Start.p, Timer.p
PackIntf.p	Packages.p
PickerIntf.p	Picker.p
SCSIIntf.p	SCSI.p
ToolIntf.p	ToolUtils.p, Events.p, Controls.p, Desk.p, Windows.p, TextEdit.p, Dialogs.p, Fonts.p, Lists.p, Menus.p, Resources.p, Scrap.p,
VideoIntf.p	Video.p

Using interface files

The interface files for the Pascal and MPW libraries as well as the Macintosh ROMs are in the {PInterfaces} directory. You can determine which interface files to use for a specific routine or data type by finding out which library or Macintosh Manager the routine or data type belongs to. You can also find out the library or Manager name by searching the {PInterfaces} directory for the routine or type name with the MPW Search command, described in the *Macintosh Programmer's Workshop 3.0 Reference.*

The compiler searches several directories for interface files, until the specified file is found. It searches the directory containing the current input file, directories specified using the `-i` option to the compiler, and directories specified in the Shell variable {PInterfaces}.

You specify the units needed for your programs by using the `uses` statement:

```
uses unitname, unitname, ... ;
```

The compiler assumes that a unit 'unitname' will be found in the file 'unitname.p'. This is the file for which it then searches. To override this assumption, use the {$U} directive. See "Compiler Directives" in Chapter 13 for details.

The form of the pathname also determines where the compiler looks for the interface file. If a *full pathname* is specified, the compiler uses exactly that name and performs no search. A full pathname contains at least one colon (:) but doesn't begin with a colon. If a *partial pathname* is specified, the compiler searches several directories for the file. Partial pathnames either begin with a colon or don't contain any colons.

Interface files can be nested up to five levels deep.

Table 1-3 summarizes the compiler's interface-file search rules.

■ **Table 1-3** Interface-file search rules

Full pathnames	
`uses` *filename*	Use the name as specified.
Partial pathnames	
`uses` *filename*	Search the following directories, in this order: 1. The directory of the source file that contains the `uses` statement 2. Directories specified by the compiler's `-i` option, in the order specified 3. Directories specified by the Shell variable `{PInterfaces}`

About the Pascal libraries

The MPW 3.0 Pascal files include several **libraries** that contain the executable object code for most of the predefined Pascal procedures and functions (described in Chapter 11) as well as the code for more specialized routines. In addition, libraries include code needed to access the Macintosh ROM routines. A full description of the Macintosh ROM routines is included in *Inside Macintosh.*

Certain libraries are shared by Pascal and one or more other languages; they are in the directory identified by the MPW 3.0 Shell variable `{Libraries}`. Three libraries (`PasLib.o`, `SANELib.o`, and `SANELib881.o`) are specific to Pascal; they are in the directory {`PLibraries`}.

Every MPW 3.0 Pascal program must be linked with the libraries `Runtime.o`, `Interface.o`, and `PasLib.o`. Others are required for different program operations, as summarized below. For further information about using these libraries, see the *Macintosh Programmer's Workshop 3.0 Reference.*

MPW Pascal includes the following libraries.

- The Standard Pascal Library in the file `{PLibraries}PasLib.o` contains all of the standard Pascal I/O routines, the heap initialization routines, and certain special I/O routines described in Chapter 10. Every MPW Pascal program must be linked with this library. The names of the special I/O routines all begin with `PL`; if you call any of them explicitly, you must use the interface file `PasLibIntf.p`. The standard Pascal I/O routines are implemented implicitly by the compiler and do not require an interface file.
- The Pascal SANE libraries in the file `{PLibraries}SANELib.o` contain the procedures and functions described in Appendix G. These procedures and functions provide accurate, extended-precision floating-point arithmetic. If you use any of them in your program, you must use the interface file `SANE.p` in your compilation and link it with the library `{PLibraries}SANELib.o`. `SANELib.o` will use the MC68881 when one is available.
- The Pascal SANE Library for the MC68881 floating-point coprocessor is included in the file `{PLibraries}SANELib881.o` and contains alternate SANE routines that call the MC68881 directly. This library does not work on machines without an MC68881.

Table 1-4 lists the library object files used with MPW Pascal. The first eight files, provided with the Macintosh Programmer's Workshop, are shared with other languages and appear in the {Libraries} directory. The remaining files, provided with MPW Pascal, are used only with Pascal and appear in the {PLibraries} directory.

■ **Table 1-4** Library object files used by MPW Pascal

Libraries that may be used with MPW Pascal	Use
`Interface.o`	*Inside Macintosh* libraries shared with other languages.
`ToolLibs.o`	Contains the code for the cursor control and manager routines described in the *Macintosh Programmer's Workshop 3.0 Reference*. If you use any of these procedures in your program, you must include the appropriate interface file in your compilation and link it with this library.
`DRVRRuntime.o`	Run-time support for desk accessories and other drivers. If your program is a desk accessory, you must link it with this library.
`ObjLib.o`	Facilities described in Chapter 12 that implement object-oriented programming without MacApp. If you use any of these techniques without using MacApp, use the interface file `ObjIntf.p` in your compilation and link it with this library.
`PerformLib.o`	Performance measurement routines. (See the *MPW 3.0 Reference* for more information on performance measurement.)
`Stubs.o`	Stubs used by the Linker to replace unused library routines for tools.
`Runtime.o`	Data initialization routines.
`HyperXLib.o`	HyperCard 'XCMD' routines
`PasLib.o`	Standard Pascal library containing all standard Pascal I/O routines and heap initialization routines.
`SANELib.o`	SANE Library of procedures and functions that provide accurate, extended-precision floating-point arithmetic.
`SANELib881.o`	SANE Library that is functionally equivalent to the library SANELib.o except this version must be used when you have invoked the `-MC68881` compiler option.

See "Linking an Application" later in this chapter for more information on using these libraries.

About the Pascal examples

The Pascal files consist of eight sample Pascal programs included with MPW Pascal: an application, a tool, a desk accessory, and a program that demonstrates the use of performance tools. In addition, the makefiles containing the commands needed to build each of the examples are provided in the same folders. These files are in {PExamples}. Table 1-5 lists these files.

■ **Table 1-5** Example source files used by MPW Pascal

Source files	PExamples folder
`Makefile`	Makefile for building sample programs
`Instructions`	Instructions for building sample programs.
`Sample.p`	Sample Pascal application. This is the sample application described in "A Simple Example Program" in Chapter 1 of *Inside Macintosh*, Volume 1. It is a simple MultiFinder-aware sample application.
`TESample`	Simple MultiFinder-aware TextEdit application.
`SillyBalls.p`	Simple color QuickDraw sample application
`TubeTest.p`	Simple color QuickDraw and Palette Manager
`ResEqual.p`	Sample application: an MPW tool.
`Memory.p`	Sample desk accessory. The Memory desk accessory displays the current free space in the application and system heaps, the free space on the default volume, and the name of the default volume. This information is updated every 5 seconds. When Memory is first opened, it calls `_MaxMem` to purge memory, thus showing the upper bounds on free space in the heaps.
`EditCdev.p`	Sample Control Panel device with a TextEdit item.
`TestPerf.p`	A sample program that uses the Pascal Performance Tools.

The file `Instructions` contains step-by-step instructions for building each of the sample programs. After installing MPW and MPW Pascal, as described in the *Macintosh Programmer's Workshop 3.0 Reference*, open this file and follow the instructions.

Installing MPW Pascal

Instructions for installing MPW Pascal on a hierarchical file system (HFS) hard disk 20 or 20SC appear in the *Macintosh Programmer's Workshop 3.0 Reference.* After installing MPW by following those instructions, run the MPW Install script and insert the MPW Pascal disk.

Alternatively, you can install Pascal with these steps:

1. Copy the file Pascal (the compiler) to the {MPW}Tools folder.
2. Copy the folder PExamples to the {MPW}Examples folder.
3. Copy the folder PInterfaces to the {MPW}Interfaces folder.
4. Copy the folder PLibraries to the {MPW}Libraries folder.

◆ *Note:* You can put the compiler, examples, and libraries in different directories, provided you change the default values of various Shell variables defined in the Startup file. You can modify the file Startup itself or, preferably, modify the file UserStartup. The following variables determine the locations of files supplied with MPW Pascal.

▪ `{Commands}`	A comma-separated list of directories containing tools and applications. The directory containing the Pascal compiler should appear in this list.
▪ `{PInterfaces}`	A comma-separated list of directories to search for PInterface files. This should include the PInterfaces directory.
▪ `{PLibraries}`	The directory containing PLibrary files. This should be the pathname of the PLibraries directory.

For more information, see the *Macintosh Programmer's Workshop 3.0 Reference.*

Segmentation control

A segment is a part of code that can be separately loaded into memory. Your program can be written without explicit segmentation or it can contain a number of different segments.

Each `'CODE'` resource in the application's resource fork corresponds to a segment containing one or more routines. (The `'CODE'` resource with ID 0 contains the jump table; other `'CODE'` resources contain routines.) At run time, a segment is automatically loaded by the Segment Loader when you call one of the routines contained in the segment. The segment is not unloaded until the application explicitly unloads it by calling `UnloadSeg`. See *Inside Macintosh* for more information about the Segment Loader.

You can specify which routines are placed in which segments in two ways. This section tells how to use the `$S` directive to specify segmentation. The *Macintosh Programmer's Workshop 3.0 Reference* explains how to use the `Link` command to modify a program's segmentation.

Segmentation helps you reduce your program's run-time memory requirements. A typical segmentation scheme divides a program into an initialization segment and a main processing segment. You can also put routines that are seldom executed—printing routines, for instance—in a separate segment that is not loaded when the program begins executing. This allows the program to be loaded faster because the printing routines are not loaded until they are needed. If you don't specify segmentation, the compiler puts the entire program into a segment called `Main`.

The `$S` directive also lets you specify several segments within a single source file. To assign source code to a segment, precede the code with a compiler directive of the form

`{$S` *segment-name*`}`

The code following this directive is placed in the named segment until the compiler reads another `$S` or the end of the source file.

- *Note:* In an `$S` directive, segment names are case sensitive. Leading spaces are not significant, and all characters are included, up to the end of the comment character.

Code for a given segment does not have to be contiguous within the source file. The program may take the following form:

```
{$S SegA}
function
{$S SegB}
function
{$S SegA}
```

and so forth. The code following an `$S` directive is placed in the named segment until the next `$S` directive is encountered or the compiler reads the end of the source file.

The compiler marks each routine with the name of its segment. Then the Linker collects all of the functions and procedures for a segment from various input files and places them into one code segment in the output file.

Creating resources

Noncode resources, such as the resources that specify menus, windows, and dialogs, can be created using the Resource Editor (ResEdit) and the Resource compiler (Rez). These tools are described in the *Macintosh Programmer's Workshop 3.0 Reference* and *the ResEdit Reference.*

Creating an application in MPW Pascal

An **application** is a program that can be run under the Macintosh Finder or MultiFinder. Applications can also be run from the MPW Shell: execution of the MPW Shell is suspended, and the application takes over the computer's memory and display while executing.

The code for an application is contained in `'CODE'` resources in the resource fork of its file. Additional resources in the same file describe the menus, windows, dialogs, strings, and other resources used by the application. *Inside Macintosh* explains in detail how to write a Macintosh application.

This section outlines the steps for building an application in MPW Pascal. The Instructions file in the PExamples folder describe some of the tools that can be used to automate the process. The MakeFile file in the PExamples folder illustrates the use of some of the tools. The *Macintosh Programmer's Workshop 3.0 Reference* describes these tools in detail.

Building an application

The easiest way to build any program in MPW is to use the Build menu. We will build Sample, an application from the Examples folder. The source files for Sample are Sample.p and Sample.r. Using the Directory menu, set the current directory to HD:MPW:Examples:PExamples.

Select Build from the Build menu and type the program name `sample`.

You will see something like this on the screen:

```
# 3:58:13 PM -----Build of Sample
# 3:58:13 PM -----Analyzing dependencies
# 3:58:14 PM -----Executing build commands
# 3:58:40 PM -----Done
  sample
```

The Build command compiles and links the application. For details on independently compiling and linking an application, see the sections "Compiling an Application"and "Linking an Application" that follow.

Press `Enter` to launch the sample application. You can cut, paste, copy, and move the cursor. Quit (Command-Q) returns you to the MPW Shell.

Compiling an application

To compile a Pascal program, first start the MPW Shell application, then enter the `Pascal` command in any window. Typically, the command specifies options and the name of the source file to the compiler, although neither is required. For example, the command

```
Pascal -p Sample.p
```

compiles the source file Sample.p, producing the object file Sample.p.o. The `-p` option specifies that progress information should be written to diagnostic output. This information appears on the screen after the command.

You can find a complete specification of the `Pascal` command—including input, output, and diagnostic specifications, status values, and options—in the *Macintosh Programmer's Workshop 3.0 Reference.*

Linking an application

The Linker is used to combine object files from several separate compilations, together with any necessary library object files, to produce the executable code resources for a program. The Linker either creates a new resource file, containing only the code resources for your program, or replaces the code resources in an existing resource file, leaving other resources, such as menus and dialogs, intact. This allows you to run the Resource compiler either before or after running the Linker. The *Macintosh Programmer's Workshop 3.0 Reference* describes the Linker in detail.

An application written partly or totally in Pascal for use on any Macintosh should be linked with the libraries listed in Table 1-6.

Link code for use on any Macintosh with these libraries:

■ **Table 1-6** Linking an application

Inside Macintosh interfaces	Run time support	Pascal libraries
{Libraries}Interface.o	{Libraries}Runtime.o {PLibraries}SANELib.o	{PLibraries}PasLib.o

Code compiled to use the MC68881 on the Macintosh II:

Inside Macintosh interfaces	Run time support	Pascal libraries
{Libraries}Interface.o	{Libraries}Runtime.o	{PLibraries}PasLib.o {PLibraries}SANELib881.o

It's wise to link new programs with all the libraries that might be appropriate. If you specify unnecessary files in the Link command, the Linker displays a message listing which files can be removed from your build instructions.

If you are using the `-mc68881` compiler option, you must place the file {PLibraries}SANELib881.o first in your link list. This file contains some definitions that override 80-bit versions in other libraries. The Linker uses the first definition it reaches, then displays warning messages when it encounters duplicate definitions. You can use the `-d` linker option to suppress warnings about duplicate definitions.

Programs written partly in Pascal and partly in assembly language or C should be linked with the file CRuntime.o and not the file Runtime.o. The Linker will detect several duplicate entry points when linking with both the Pascal and the C libraries. All but one of these duplicates can be safely ignored: the copies of the routines are identical.

The exception is the execution starting point. If execution is expected to begin with the C function `main()`, no special precautions are necessary. However, if your main program is written in assembly language or Pascal but parts of your program are written in C (and must therefore be linked with file CRuntime.o), the object file containing your main program must appear before CRuntime.o in the list of object files passed to the Linker.

Creating a tool in MPW Pascal

A **tool** is a program that operates within the MPW Shell environment. The Pascal compiler, Rez, and Link are all tools. You can write your own tools in Pascal, C, or assembly language. The *Macintosh Programmer's Workshop 3.0 Reference* describes tools and how they are created. This section contains specific information about writing tools in Pascal.

You execute a tool by entering an MPW command. The parameters specified in the command line are passed as parameters to the main program. The Shell variables that are exported are also passed as a parameter to the main program; they can be accessed directly or by using the `getenv()` function from the Pascal Library. To access these parameters, use interfaces as follows:

```
USES
 CursorCtl,
 IntEnv,
 PasLibIntf;
```

You can find additional details about parameters to tools in the *Macintosh Programmer's Workshop 3.0 Reference.*

Tools have direct access to MPW Shell windows and selections. The `FILE` variables `stdin`, `stdout`, and `stderr` refer to MPW's standard input, standard output, and diagnostic output, respectively. By default, Pascal Library I/O functions read standard input (text entered from the Shell) and write to standard Pascal output. Any files opened by tools, using either Pascal Library functions or *Inside Macintosh* library functions, read and write to windows if the file specified is open in a window. The contents of the window are read or written in place of the data fork of the file. Selections in windows can also be read and written as if they were files, by adding the suffix .§ to the filename (for example, HD:MPW:Worksheet.§).

Building a tool

The easiest way to build any program in MPW is to use the Build menu. We will build ResEqual, a sample MPW tool that compares the resources in two files. The source files for ResEqual are ResEqual.p and ResEqual.r; since a makefile already exists, you don't need to create one. Using the Directory menu, set the current directory to HD:MPW:Examples:PExamples.

Now select Build from the Build menu and type the program name ResEqual.

You will see something like this on the screen:

```
# 10:58:07 PM ----- Build of ResEqual.
# 10:58:08 PM ----- Analyzing dependencies.
# 10:58:10 PM ----- Executing build commands.
  Rez :Examples:PExamples:ResEqual.r -append -o ResEqual
  Pascal :Examples:PExamples:ResEqual.p
  Link -w -t MPST -c 'MPS ' "Oya:.MPW:Libraries:"Runtime.o
"Oya:.MPW:Libraries:"Interface.o "Oya:.MPW:PLibraries:"PasLib.o
"Oya:.MPW:PLibraries:"SANELib.o "Oya:.MPW:Libraries:"ToolLibs.o
:Examples:PExamples:ResEqual.p.o -o ResEqual
# 10:58:35 PM ----- Done.
  ResEqual
```

Now press `Enter`.

Compiling a tool

You compile a tool in exactly the same way you compile an application. The previous information regarding include-file search rules, segmentation, and resources applies equally to tools and applications.

Linking a tool

The MPW Shell recognizes a tool by the type and creator. Specify the following options when linking a tool:

```
Link -t MPST -c "MPS " ...
```

This command specifies the file type and creator of an MPW tool. Follow the same library linking rules for tools as for applications (see the section "Linking an Application"). In addition, if your tool calls any of the spinning cursor or error manager routines, link with the following libraries:

```
{Libraries}Stubs.o
{Libraries}ToolLibs.o
```

The file `Stubs.o` contains a collection of "stubs," or dummy routines, for several functions that are defined in the run-time library but are not necessary for MPW tools running under the MPW Shell. You can use these stubs to reduce the size of a tool. `Stubs.o` should be linked in before any of the other libraries.

Creating a desk accessory in MPW Pascal

A **desk accessory** is a program that you run by selecting it from the Apple menu. It shares its execution environment with the currently executing application. Information on writing desk accessories appears in the Desk Manager and Device Manager chapters of *Inside Macintosh* and in the *Macintosh Programmer's Workshop 3.0 Reference*. This section contains information specific to writing desk accessories in MPW Pascal.

Desk accessory restrictions

A desk accessory has neither a jump table nor a global data area.

- Because it does not have a jump table, a desk accessory must be in a single segment. Either omit segmentation specifications so that all your code is placed in the default segment, or use identical segmentation specifications for all of your routines. Use the Link command to move any library routines you use into your single segment.
- Because it does not have a global data area, a desk accessory written in Pascal must not use global variables. Furthermore, a desk accessory cannot call library routines that require global data. Programming hints for avoiding these restrictions appear in the *Macintosh Programmer's Workshop 3.0 Reference.*

The DRVRRuntime library

Desk accessories have traditionally been written in assembly-language source, partly because of the peculiar resource format used by the system for desk accessories, the `'DRVR'` resource. Setting up the `'DRVR'` layout header, passing register-based procedure parameters, and coping with the nonstandard exit conventions of the driver routines have made it fairly difficult in the past for programmers not familiar with assembly language to implement desk accessories in higher-level languages.

To overcome these difficulties and simplify the task of writing a desk accessory in Pascal, MPW provides the library DRVRRuntime.o and the resource type `'DRVW'` declared in MPWTypes.r. Together they compose the driver layout header and the five entry points that set up the `open`, `prime`, `status`, `control`, and `close` functions of a driver. For more information about `'DRVR'` resources, see the Device Driver chapter of *Inside Macintosh*, Volume 2. For an example defining desk accessory resources, see the file Memory.r in the folder PExamples.

Using the library DRVRRuntime.o to create desk accessories offers a number of advantages:

- No assembly-language source is required. Each of the driver routines—`DRVROpen`, `DRVRPrime`, `DRVRStatus`, `DRVRControl`, and `DRVRClose`—can be written in Pascal.
- The DRVRRuntime library handles desk accessory exit conventions: your routines simply return a result code.

The DRVRRuntime library consists of a main entry point that overrides the Pascal run-time initial entry point. The DRVRRuntime entry point contains driver "glue" that sets up the parameters for you, calls your routine, and performs the special exit code required by a desk accessory to return control to the system. Your routines perform the actions of the desk accessory, such as opening a window or responding to mouse clicks in it.

Desk accessory routines

Desk accessories that use the library DRVRRuntime must contain the five functions `DRVROpen`, `DRVRPrime`, `DRVRStatus`, `DRVRControl`, and `DRVRClose`. All of these functions have the same parameter and result types. They are declared as Pascal-compatible functions so that the library DRVRRuntime can be used for writing desk accessories in Pascal, C, and assembly language. Each of these five routines should be declared as follows:

```
FUNCTION DRVROpen(ctlPB: ParmBlkPtr; dCtl: DCtlPtr): OSErr;
BEGIN
      ... your code ...
      DRVROpen := resultCode;
END;
```

Types `ParmBlkPtr` and `DCtlPtr` are defined in the Files.p file. Type `OSErr` is defined in MemTypes.p. Details on each function appear in the *Macintosh Programmer's Workshop 3.0 Reference*, in the Desk Manager chapter of *Inside Macintosh*, Volume 1, and in the Device Manager chapter of *Inside Macintosh*, Volume 2.

Building a desk accessory

The easiest way to build any program in MPW is to use the Build menu. We will build Memory, a sample desk accessory that displays the memory available in the application and system heaps, and on the boot disk.

The source files for Memory are Memory.c and Memory.r; since a makefile already exists, you don't need to create one. Using the Directory menu, set the current directory to HD:MPW:Examples:PExamples.

Using the Directory menu, set the directory to `HD:MPW:Examples:PExamples`. Now select Build from the Build menu and type the program name Memory. You will see something like this on the screen:

```
# 4:12:40 PM ----- Build of Memory.
# 4:12:41 PM ----- Analyzing dependencies.
# 4:12:43 PM ----- Executing build commands.
  pascal Memory.c
Link -w -rt DRVW=0 -sg Memory "Oya:MPW:Libraries"DRVRRuntime.o
Memory.p.o
"Oya:MPW:Libraries"Interface.o "Oya:MPW:PLibraries"Paslib.o -o
Memory.DRVW -c "????" -t "????"
  Rez -rd -c DMOV -t DFIL Memory.r -o Memory
# 4:13:06 PM ----- Done.
  'Font/DA Mover' 'Oya:System Folder:System' Memory # Install DA
```

Press `Enter` to launch the Font/DA Mover. (If you have two megabytes or less of RAM, you may not be able to do this under MultiFinder; restart with the Command key held down, then try again.) Install the Memory DA in your System file. It gives you the current size of the System Heap and the Application Heap.

Linking a desk accessory

A desk accessory written in Pascal must be linked with both `DRVRRuntime.o` and `Runtime.o`. `DRVRRuntime.o` must precede `Runtime.o` in the list of object files passed to the Linker, for example

```
LINK {Libraries}DRVRRuntime.o        {Libraries}Runtime.o
```

Creating code for different models of the Macintosh

Using version 3.0 of MPW Pascal, you can create applications that run on all models of the Macintosh. This section outlines the compatibilities among the machines and the strategies for writing and compiling code that will run on the different models. Two megabytes of RAM are required.

Source code

You can write your source code to be compatible with one or more models of the Macintosh. You have four primary options:

- Code written for a Macintosh 512K also runs on a Macintosh XL, a Macintosh Plus, a Macintosh SE, and a Macintosh II. If you want your program to run on any model, follow the recommendations in *Inside Macintosh*, Volumes 1 through 3.
- Code written for a Macintosh Plus also runs on a Macintosh SE and a Macintosh II. If you want your program to run on either of these models, follow the recommendations in *Inside Macintosh*, Volumes 1 through 4.
- Code written for a Macintosh SE also runs on a Macintosh II and a Macintosh 512Ke or Macintosh Plus. If you want your code to run on any model with the most recent system disk, follow the recommendations for a Macintosh SE in *Inside Macintosh*, Volume 5.
- Code written for a Macintosh II, using the ROM code that is present only in that model, runs only on a Macintosh II.

Interface files

A set of interface files provided with MPW Pascal gives you access from Pascal to the Macintosh Toolbox and Macintosh Operating System routines built into the Macintosh ROMs. Volume 5 of *Inside Macintosh* describes the ROM code that is new with the Macintosh II and the Macintosh SE. Volume 4 of *Inside Macintosh* describes the code for the Macintosh Plus.

Much of the new material is usable only on a Macintosh II, because it makes use of hardware options that are not available on other models. You can include all of the interface-file definitions in code for use on any model, but you cannot call ROM routines that are not present on the machine that will run the compiled code.

See *Inside Macintosh*, Volume 5, for detailed descriptions of the new material and the models with which it can be used.

Compiler options

With the addition of the Macintosh II to the product line, there are now compiler differences among the models as well as ROM code differences. These compiler differences are discussed in the following sections.

SANE and the Macintosh II

MPW Pascal includes numeric capabilities that conform to the Institute of Electrical and Electronics Engineers (IEEE) Standard 754 for Floating-Point Arithmetic. This Standard is the set of guidelines defined by the IEEE for the design and implementation of systems that perform floating-point arithmetic.

The Standard Apple Numeric Environment (SANE) is Apple's implementation of these guidelines. MPW 3.0 Pascal uses SANE to provide a powerful, flexible environment for numeric calculations.

The IEEE Standard recommends the implementation of two additional data types for numeric programming, in addition to the `real` type that's specified in the ANSI Standard. MPW Pascal includes these two additional types. They are described in Appendix G. You'll also find references to SANE in the descriptions of predefined arithmetic functions in Chapter 11.

SANE includes a library (SANELib) of useful numeric procedures and functions. This library, described in Appendix G, works on all machines in the Macintosh family.

Floating-point enhancements

Applications using the SANE packages (`Pack4` and `Pack5`) run faster on the Macintosh II because of the 68881 floating-point coprocessor. The default mode of the compiler is to call these packages for all floating-point operations. For the fastest possible arithmetic on machines with a 68881, the compiler has an option that forces direct calls to the 68881. When the option is used, the resulting code will not run on Macintoshes without both a 68020 and a 68881.

The SANE interface has been extended to provide support for the 68881: one new constant for setting the default environment to work in both the 68000 and the 68881 worlds, two new functions for transferring between extended formats, and two new functions for access to the 68881 trap mechanism. The code for these new features is included in a new library SANELib881.o. The features are described in the updated interface file, SANE.p, and are discussed in detail in "Converting Between Extended Formats in Mixed-World Programs," in Appendix G.

The `SetEnvironment(0)` call will not work under the `-MC68881` option. Replace it with `SetEnvironment(IEEEDefaultEnv)`, which works with or without the `-MC68881` option.

MC68881 enhancements

The Motorola 68881 does basic arithmetic and a large number of transcendental functions very fast. Ordinarily, the MPW Pascal compiler generates calls to the SANE packages (`Pack4` and `Pack5`) for floating-point operations; if a 68881 is present, the SANE packages use it, so floating-point packages are automatically faster. To take better advantage of the 68881, the Pascal compiler has been modified to provide optional direct calls to the coprocessor.

To access the 68881 directly for greater speed in basic arithmetic calls, type `-MC68881` on the command line or use the equivalent `$MC68881+` compiler directive (described in detail in Chapter 13) and link with SANELib881.o instead of SANELib.o. With the `-MC68881` option, the `extended` type is 12 bytes long and variables of the `extended` type may be allocated to registers.

The `-MC68881` option will result in the use of transcendental functions whose accuracy is identical to that of the SANE packages. For the faster but less accurate transcendental functions provided on the 68881, type `-d Elems881=true` on the command line. For details, see Chapter 13.

For details on using the `-MC68881` option, see Appendix G.

△ **Important** Use of the `-MC68881` option can generate instructions incompatible with the 68000. Your program might not run on a Macintosh without the 68881. △

MC68020 enhancements

MPW Pascal supports the Motorola 68020 central processing unit with the compiler option -MC68020 or the equivalent compiler directive $MC68020+ (described in Chapter 13). The 68020 yields faster longint arithmetic and improved performance with packed structures. See Appendix K for advanced programming techniques for the MC68020.

△ **Important** Use of the -MC68020 option can generate instructions incompatible with the 68000. △

Other MPW 3.0 products

The MPW 3.0 Shell provides an integrated working environment within which you can write programs in assembly language, Pascal, and C.

If you write programs in Pascal, you can also use MacApp for object-oriented programming, as described in Chapter 12. The MPW 3.0 Pascal compiler cannot be used as a stand-alone program.

Besides the assembly-language Assembler and the Pascal and C compilers, the MPW 3.0 Shell also contains a rich complement of editing, linking, and debugging tools including SADE and MacsBug.

Here's how to find more information about the ways you can combine your Pascal programs with other MPW 3.0 facilities:

- For more information about the MPW 3.0 Shell, including how to edit your source text, how to use the Build Menu facility, how to use Commando, how to use the command language, how to create resources for your program, and how to use the linking and debugging tools, consult the *Macintosh Programmer's Workshop 3.0 Reference.*
- If you want to use assembly-language subroutines in your Pascal programs, or vice versa, consult the *Macintosh Programmer's Workshop 3.0 Assembler Reference.*
- If you want to write your program partly in Pascal and partly in C, consult the *Macintosh Programmer's Workshop C 3.0 Reference* and Appendix E of this reference.
- If you want to use MacApp to write object-oriented programs in MPW 3.0 Pascal, consult the *MacApp 2.0 Programmer's Reference* and Chapter 12 of this reference.

Many of the books just cited are all listed under "Other Reference Materials" in the Preface.

Chapter 2 **Symbols**

THIS CHAPTER DISCUSSES SYMBOLS, the smallest meaningful units of source text in a Pascal program. ■

Contents

Symbols

This chapter discusses symbols under the following headings:

- special symbols and reserved words
- identifiers
- numbers
- labels
- quoted string constants
- delimiters
- directives
- comments and Compiler directives

Every Pascal source text consists of a succession of such symbols. You write each symbol as a string of ASCII characters, according to these rules:

- Each symbol must be complete and unbroken; you may not insert one symbol within another.
- You can write comments anywhere, as long as they do not break up other symbols. The Pascal Compiler simply skips over them.
- Subject to certain exceptions, explained below, you must write delimiters alternately with the other symbols. The Compiler uses delimiters to determine where other symbols begin and end.

The character set used by MPW Pascal is eight-bit extended ASCII, with characters represented by numeric codes in the range 0..255. See Appendix C for the complete character set.

The Compiler does not recognize the ASCII control codes (ASCII 0 through ASCII 31), except tab and carriage return. Otherwise, it processes the following subsets of the ASCII character set:

- The **letters** are those of the English alphabet, *A* through *Z* and *a* through *z*.
- The **digits** are the Arabic numerals 0 through 9.
- The **hex digits** are the Arabic numerals 0 through 9, the letters *A* through *F*, and the letters *a* through *f*.
- The **blanks** are the space character (ASCII 32), the horizontal tab character (ASCII 9), the return character (ASCII 13), and option-space (ASCII 202).
- The **underscore,** ASCII 95.

Special symbols and reserved words

Special symbols and **reserved words** are symbols having fixed meanings. If you try to change their meanings or use them in ways other than their intended uses, the Compiler will issue an error. The following single characters are special symbols:

```
+  -  *  /  =  <  >  [  ]  .  ,  (  )  :  ;  ^  @  {  }  $  &  |
```

The following character pairs are special symbols:

```
<>   <=   >=   :=   ..   (*   *)   (.   .)   **
```

Some of the special symbols are also operators. Operators are defined in Chapter 3.

◆ *Note:* The symbols (. and .) are equivalent to [and].

The reserved words in MPW Pascal are listed in Table 2-1.

■ **Table 2-1** Reserved words

AND	DOWNTO	IF	NIL	PROGRAM	TYPE
ARRAY	ELSE	IMPLEMENTATION	NOT	RECORD	UNIT
BEGIN	END	IN	OF	REPEAT	UNTIL
CASE	FILE	INTERFACE	OR	SET	USES
CONST	FOR	INTRINSIC*	OTHERWISE	STRING	VAR
DIV	FUNCTION	LABEL	PACKED	THEN	WHILE
DO	GOTO	MOD	PROCEDURE	TO	WITH

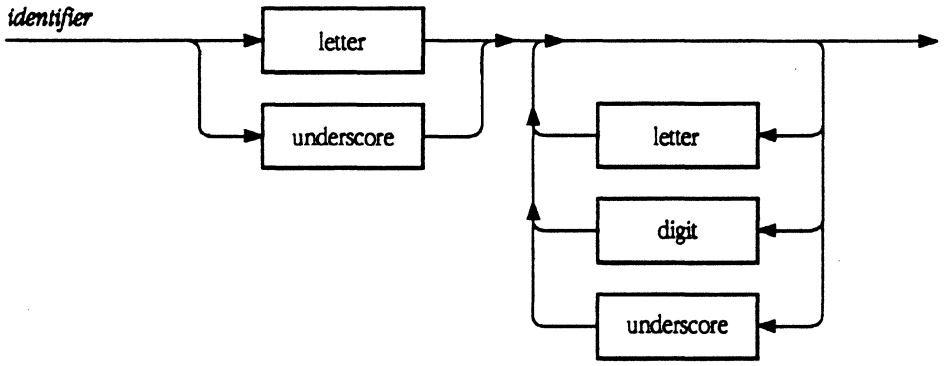
* INTRINSIC is reserved for future use.

These reserved words appear in uppercase letters throughout this book. However, MPW Pascal is not case sensitive—corresponding uppercase and lowercase letters are equivalent.

Identifiers

Identifiers are the names that denote constants, types, variables, procedures, functions, units and programs, and fields in records. Here are the rules for writing identifiers:

- An identifier can be of any length, but only the first 63 characters are significant.
- They are not case sensitive; corresponding uppercase and lowercase letters are equivalent.
- They may contain only letters, digits, and underscore characters (ASCII 95); in particular, they may not contain spaces.
- Every identifier must begin with a letter or an underscore.

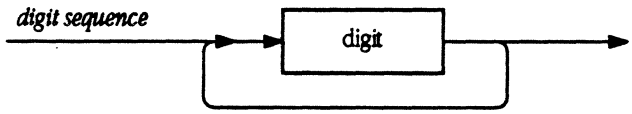


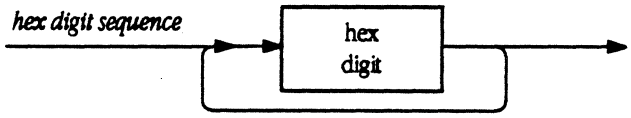
Here are some examples of identifiers:

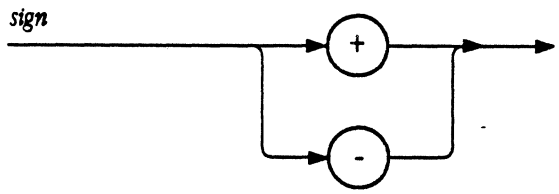
```
Z     Knowledge     SUM     get_byte
An_identifier_can_be_as_long_as_you_want_stop
```

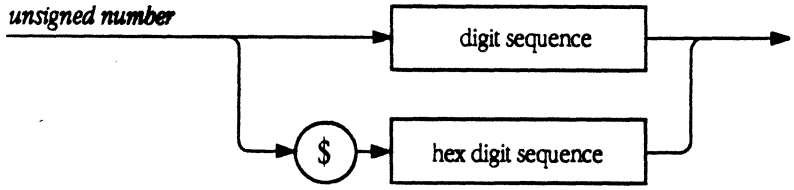
Numbers

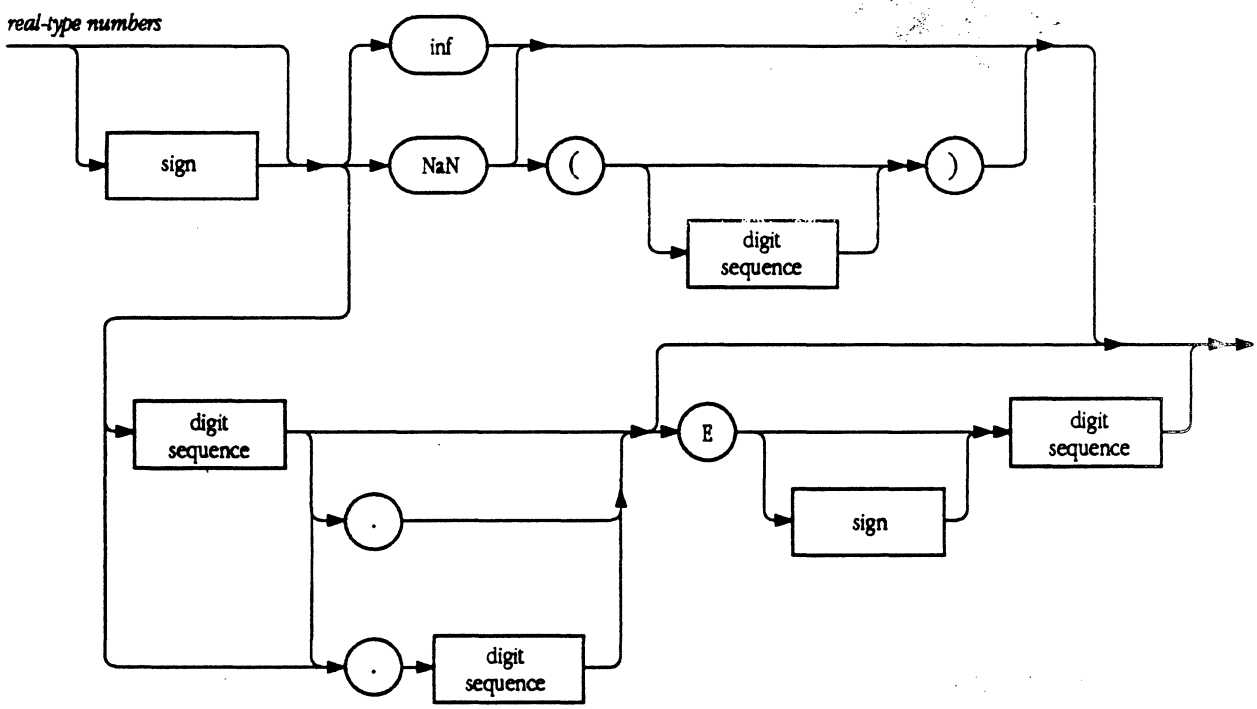
Within an MPW Pascal program, you can use ordinary decimal notation for numbers that are constants of the data types `integer` and `longint` and the real types (see Chapter 4). You can also write hexadecimal integer constants using the $ character as a prefix. Finally, you can use scientific notation (`E` or `e` followed by an exponent) for real types. Here are the syntax diagrams for writing numbers:











The letter `E` or `e` preceding the scale factor in an unsigned real means "times ten to the power of."

These are examples of correct notation for numbers in MPW Pascal programs:

```
1       +100       -0.1       5E-3       87.35e+8       $A05D
```

Notice that `5E-3` means 5×10^{-3} and `87.35e+8` means 87.35×10^{8}. You can omit the plus sign (+) before the exponent so that `8E+7` and `8E7` are equivalent.

Numbers written with a decimal point or exponent are stored as type `extended` (unless explicitly assigned to a variable of another of the real types). Other decimal numbers are stored as the smallest numerical type (`integer` or `longint`) needed for that value. For example, an `integer` value from –32768 to 32767 is stored in two bytes, as type `integer`. See Chapter 4 for the value limits of different numerical data types.

A hexadecimal constant with one to four digits is stored as an `integer` (two-byte) quantity; one with five to eight digits is stored as a `longint` (four-byte) quantity. An integral hexadecimal value with more than eight significant digits causes an overflow error. Leading zeros are counted in hexadecimal digit counts. The sign of the resulting value is implied by the hexadecimal notation.

Here are some examples of hexadecimal constants and their integer values:

```
$F=15
$FFFF=-1 (integer)
$0FFFF=65535 (longint)
$FFFFF=1048575
$FFFFFFFF=-1 (longint)
```

Labels

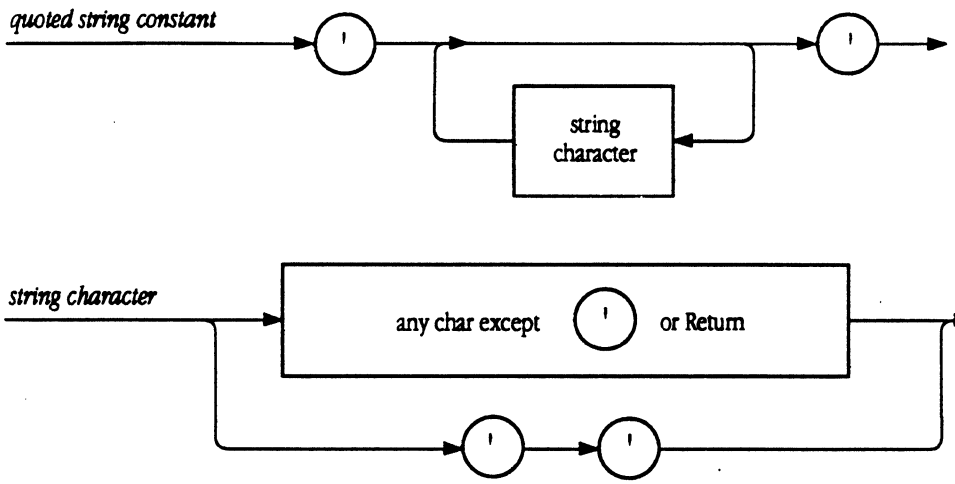
A **label** is a digit sequence in the range 0..9999. Leading zeros are not significant in labels. For example, `0078` and `78` are equivalent.

Labels are used with `GOTO` statements, described in Chapter 7.

Quoted string constants

A **quoted string constant** is a sequence of zero or more characters from the ASCII character set given in Appendix C. Here are the rules for writing quoted string constants:

- Each constant must be written all on one line of the program source text.
- Each must be enclosed by single quotation marks (apostrophes).
- Blanks count as characters in quoted string constants.
- The maximum number of characters in one constant is 255.
- A quoted string constant with nothing between the single quotation marks denotes the null string.
- If you want the quoted string constant to contain a single quotation mark, you must write the single quotation mark twice.



These are examples of quoted string constants:

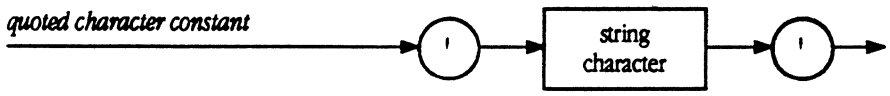
```
'Baltic'    'NOVOGOROD'       'Don''t Panic!'
'A'           ';'           ''''            ''
```

The last is a **null string.** The next to last contains one single quotation mark.

All string values have a length attribute (see "String Types" in Chapter 4). In the case of a quoted string constant, the length is fixed; it is equal to the actual number of characters in the string value.

Quoted character constants

Syntactically, a quoted character constant is simply a quoted string constant whose length is exactly one. `'A'` is an example of a quoted character constant.



A quoted character constant is compatible with any `char` type or `STRING` type; that is, it can be used either as a character value or as a string value.

Delimiters

Delimiters are symbols that separate other symbols in the source text so that the Compiler can distinguish them as discrete objects. Blanks (spaces, tabs, carriage returns, and option-spaces) are the principal delimiters. In addition, all the special symbols listed earlier in this chapter serve as delimiters while performing their other functions. Hence the Compiler can process the expression

```
2+seven=number_of_planets
```

even though it contains no spaces or tabs, because + and = are delimiters.

Comments and Compiler directives (described below) also act as delimiters.

Directives

Directives are words that have special meanings only when used in place of a procedure or function block. They are not reserved and can be used as identifiers in other contexts. FORWARD, EXTERNAL, C, and INLINE are the four directives used by MPW Pascal. INLINE is different from the other three in that it is followed by a list of constants, which make up a machine-language subprogram used by the Compiler in interpreting the directive. See Chapter 8 for more information about INLINE.

Special directives for Object Pascal

The words INHERITED and SELF have special meanings only when used in an Object Pascal method declaration (discussed in Chapter 12). You can use the words *inherited* and *self* as identifiers anywhere but within a method. In practice, Object Pascal programs consist almost entirely of methods, so INHERITED and SELF are rarely used as identifiers in Object Pascal programs. Ordinary Pascal programs never contain methods.

The word OVERRIDE is used like a directive. It has special meaning only when used after a method heading in an object type declaration. However, OVERRIDE is added to the method and does not replace its block.

Comments and Compiler directives

The constructs

```
{ any text not containing right-brace }
(* any text not containing star-right-paren *)
```

are called **comments.** They are ignored by the Compiler.

A comment cannot be nested within another comment formed with the same kind of delimiters. However, a comment formed with `{ . . . }` delimiters can be nested within a comment formed with `(* . . . *)` delimiters, and vice versa.

- *Note:* The use of nested comments is one of the differences between MPW Pascal and ANS Pascal. Nested comment structures allow you to "comment out" source text that contains only one style of comment delimiters—that is, render it invisible to the Compiler.

A **Compiler directive** is a comment that contains a `$` character immediately after the `{` or `(*` that begins the comment. The `$` character is followed by the mnemonic of the Compiler command. Compiler directives are similar to the **Compiler options** you enter through the MPW Pascal command line, the main difference being that you embed directives in the source text of your program. They are listed in Chapter 13.

Chapter 3 **Blocks and Scope**

THE BLOCK IS THE FUNDAMENTAL UNIT of Pascal source text. Each block is part of one of the following listed items:

- a procedure declaration
- a function declaration
- a program
- a unit

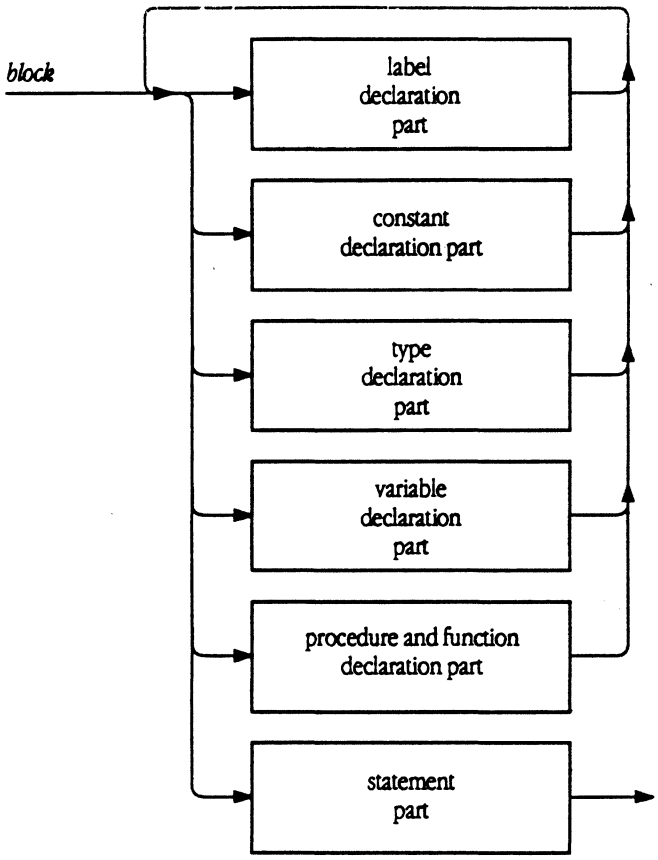
Each block consists of declarations and statements, constructed according to these rules:

- No specific declaration parts are required.
- Declarations may be written or intermixed in any order. ■

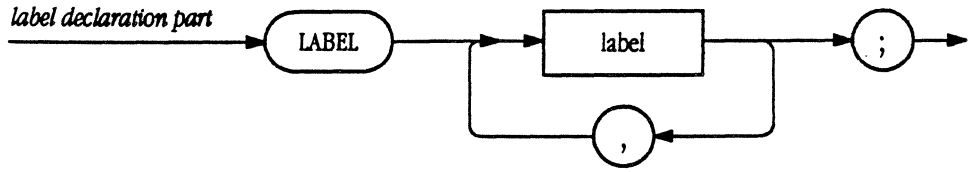
Contents

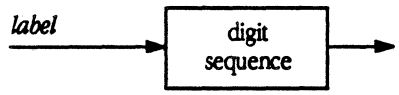
Block syntax

The following diagrams specify the overall syntax of any block:



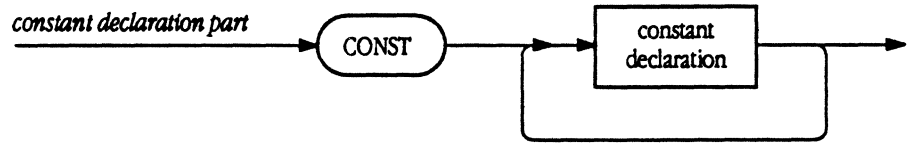
The **label declaration part** declares all labels that mark statements in the corresponding statement part. Each label must mark exactly one statement in the statement part.



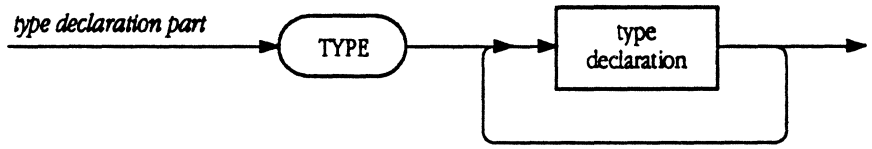


The digit sequence used for a label must be in the range 0..9999.

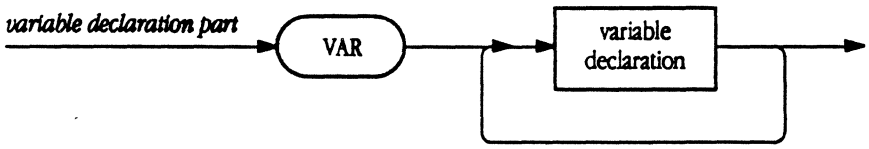
The **constant declaration part** contains all constant declarations local to the block.



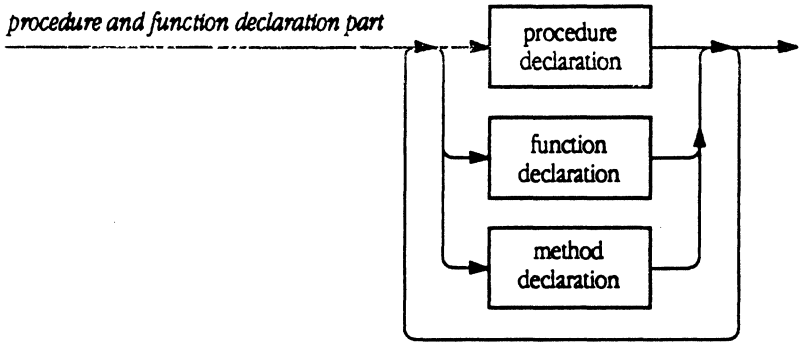
The **type declaration part** contains all type declarations local to the block.



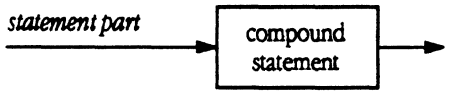
The **variable declaration part** contains all variable declarations local to the block.



The **procedure and function declaration part** contains all procedure and function declarations local to the block.



The **statement part** specifies the actions to be executed by the block.



- *Note*: At run time, all variables except file variables declared within a particular block have unspecified values each time the statement part of the block is entered. File variables are initialized to NIL.

The next section discusses the scope of items within the program or unit in which they are defined. See Chapter 8 for the scope of items defined in the interface part of a unit and referred to in a host program or unit.

Scope rules

The appearance of an identifier or label in a declaration defines the identifier or label. All subsequent occurrences of the identifier or label must be within the **scope** of its declaration.

Ordinarily, the scope of an identifier or label extends from its declaration onward to the end of the current block, including all blocks enclosed by the current block within that area. There are several exceptions to this rule, however. They are explained below.

◆ *Note:* Additional anomalies in the MPW Pascal scope rules are described in Appendix B.

Redeclaration in an enclosed block

Suppose that `Outer` is a block and `Inner` is another block that is enclosed within `Outer`. If an identifier declared in block `Outer` has a further declaration in block `Inner`, then block `Inner` and all blocks enclosed by `Inner` are excluded from the scope of the declaration in block `Outer`.

Object identifiers cannot be redeclared.

Position of declaration within its block

The declaration of an identifier or label must precede all corresponding occurrences of that identifier or label in the program text. In other words, identifiers and labels cannot be used until after they are declared. However, there are two exceptions to this rule:

- The base type of a pointer type can be an identifier that has not yet been declared. In this case, the identifier must be declared somewhere in the same type declaration part in which the pointer type occurs.
- An object type identifier may appear before it is declared, as long as that appearance is in the same type declaration part as the declaration.

Redeclaration within a block

An identifier or label cannot be declared more than once in the outer level of a particular block, except for record and object field identifiers.

A record field identifier is declared within a record type. It is meaningful only in combination with a reference to a variable of that record type. Therefore, the following redeclarations are possible:

- A field identifier can be redeclared within the same block, as long as it is not declared again at the same level within the same record type.
- An identifier that has been declared to denote a constant can be redeclared as a record field identifier in the same block.

Declarations in units

Identifiers declared in the interface part of a unit have a scope that extends to the end of a unit. The scope of these identifiers also extends to include any other units or programs that reference the unit in a `USES` clause.

Identifiers declared in the implementation part of a unit have a scope that extends to the end of the unit. These identifiers are hidden from any other units or programs that reference the unit in a `USES` clause.

For a more complete discussion of units, the interface part, and the implementation part, see "Unit Syntax" in Chapter 9.

Predefined identifiers

MPW Pascal provides a set of predefined constants, types, procedures, and functions. The identifiers of these objects, along with the statement identifiers `Cycle` and `Leave`, behave as if they were declared in a "super-outermost" block enclosing the entire program; thus, their scope includes the entire program.

Special rule for object types

In addition to having normal identifier scope, the scope of any object type identifier, object field identifier, or method identifier extends over the following areas:

- all descendants of its type
- all procedure and function blocks that implement methods of that object type and its descendants

The following extra redeclaration rules apply to object types and their associated identifiers:

- If you declare the identifier OBJECT in a program that uses Object Pascal, the Compiler will issue an error.
- Object field identifiers can be redeclared in objects that are not descendants of the original object type. However, they cannot be redeclared in any descendant of the object type where they are originally declared, even if that object is declared in a different block.
- Method identifiers can be redeclared, but the parameter list and return value (if any) for the new method must be identical to those for the original method.

Scopes, object files, and other languages

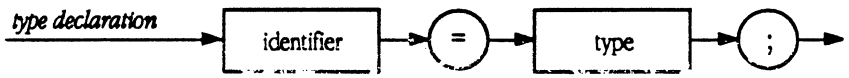
The discussion of scopes in this chapter assumes that programs are written entirely in Pascal. Pascal provides strong type checking at compile time and a secure mechanism (the unit) for sharing global declarations across modular compilations. Other languages, such as C and assembly language, do not have identical mechanisms. To mix Pascal with other languages, you may need to use some of the Compiler options that modify the default treatment of Pascal symbols in object files. See the discussion of the `$N+`, `$Z*`, and `$Z+` options in Chapter 13.

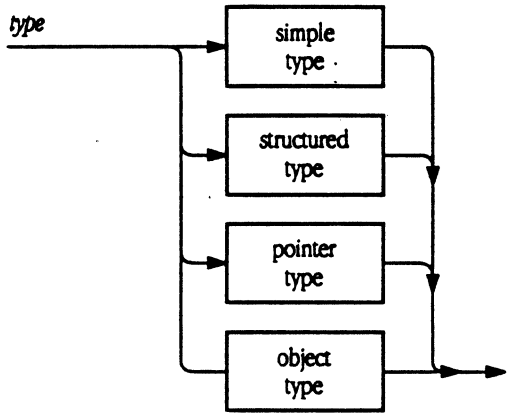
Chapter 4 Data Types

YOU MUST SPECIFY A **TYPE** when you declare a variable. The type determines the set of values that variable can assume and the operations that can be performed upon the variable. ■

Contents

A **type declaration** associates an identifier with a type.





The occurrence of an identifier on the left side of a type declaration declares it as a type identifier for the block in which the type declaration occurs. The scope of a type identifier does not include its own declaration, except for pointer types and object types. The MPW Pascal data types are arranged as shown in Table 4-1.

■ **Table 4-1** Data types

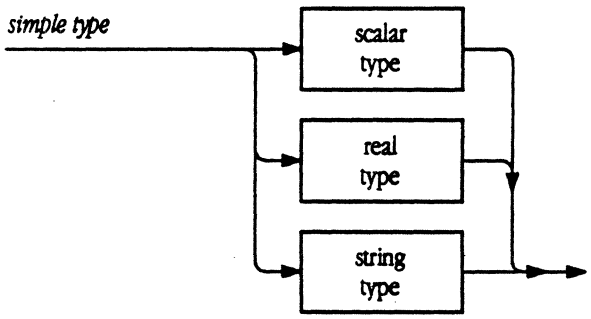
Simple types	Pointer type	Structured types
Real types		`ARRAY`
`real`, `single`*		`RECORD`
`double`*		`SET`
`extended`*		`FILE`
`comp`, `computational`*		`OBJECT`
Scalar types		
`integer`*		
`longint`*		
`char`*		
`boolean`*		
enumerated types		
subrange types		
String types*		

The types marked with an asterisk in Table 4-1 are **predefined;** their type declarations are built into the Compiler. Others are **user-defined** and require a prior type declaration in your source text.

The types listed in Table 4-1 are discussed in the rest of this chapter.

Simple types

All the simple types define ordered sets of values.



The **simple types** include real types, scalar types, and strings.

Real types

There are four real types in MPW Pascal, all predefined. They are listed in Table 4-2.

■ **Table 4-2** Real types

Identifiers	Values	Memory size
`real, single`	floating-point numbers	4 bytes
`double`	floating-point numbers	8 bytes
`extended`	floating-point numbers	10 bytes (without `-MC68881`)
`extended`	floating-point numbers	12 bytes (with `-MC68881`)*
`comp, computational`	whole numbers	8 bytes

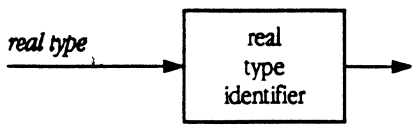
**With -68881, all floating point types use 12 bytes.*

All the floating-point calculations required in MPW Pascal programs are performed according to the specifications for the Standard Apple Numeric Environment (SANE). SANE is based on the IEEE Standard for Floating-Point Arithmetic, which recommends the use of four floating-point types in high-level languages. In the IEEE Standard, the types are called `single`, `double`, `extended`, and `comp`. SANE provides the three additional floating-point types included in MPW Pascal.

The numeric environment for the real types uses IEEE Standard defaults: numbers are rounded to the nearest value in extended precision, and all halts are disabled. Each program begins with these defaults and with all exception flags clear. Functions for managing the environment and changing these parameters are included in the SANE library, which is discussed in Appendix G of this manual.

The ANS `real` type is identical to the SANE `single` type. The MPW Compiler will accept both identifiers and treats them identically. In addition, the Compiler treats the names `comp` and `computational` in exactly the same way.

The real types are written as follows:



These are the possible values for real-type variables:

- Finite values (a subset of the mathematical real numbers). As constants, these values can be denoted as described under "Numbers" in Chapter 2. The value zero has a sign, like other numbers, which appears in textual output.
- Infinite values, `+INF` and `-INF`. These arise either as the result of an operation that overflows its intended storage type or as the result of dividing a finite value by zero.
- NaNs (the word *NaN* stands for Not a Number). NaNs arise as the result of operations that have no meaningful numeric result. For example, the result of multiplying ∞ by zero is a NaN. In textual output, a NaN appears as `NAN`, followed by a set of parentheses enclosing an integer that identifies the source of the NaN.

The four real types differ in the range and precision of values that they can hold and in the amount of storage space they require:

- `Real` (or `single`) type variables take up four bytes of storage. The magnitude of `real` type values can range from approximately 1.401298464E–45 to 3.402823466E38 in scientific notation. They have 7 to 8 digits of precision.
- `Double` type variables take up eight bytes of storage. The magnitude of `double` values can range from approximately 5.0E–324 to 1.7E308 in scientific notation. They have 15 to 16 digits of precision.
- `Extended` type variables take up ten bytes of storage (12 bytes with the -MC68881's flag). The magnitude of `extended` type values can range from approximately 1.9E–4951 to 1.1E4932 in scientific notation. They have 19 to 20 digits of precision.
- The `comp`, or `computational`, data type holds only `integer` values in the approximate range ±9.2E18. (The exact range is $-2^{63}+1$ to $2^{63}-1$; -2^{63} is treated as a NaN.) `Comp` type variables are used for fixed-point values, where the decimal point is placed by the application. Although `comp` values appear to be more like the integer types than like the other real types, computations using `comp` values are performed as with the real types. `Comp` values are converted to `extended` before computations are performed.

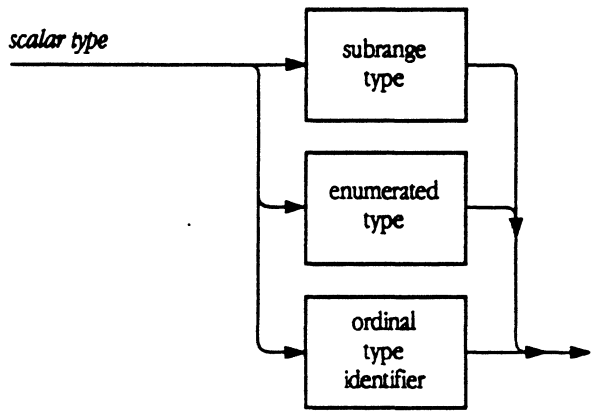
◆ *Note*: `Real` values are converted to `extended` before calculations are performed, so calculations using the `extended` data type are faster and more compact than other real-type calculations. You may want to declare all real-type temporary variables, formal value parameters, and function results as `extended` in order to save execution time and code size. External data should be stored as one of the smaller types rather than as `extended`, which varies among SANE implementations.

Scalar types

Scalar types are simple types with the following special characteristics:

- Within a given scalar type, all possible values form an ordered set and each possible value is associated with an ordinality, which is an `integer` or `longint` value. Except for `integer` and `longint` values, the first value of the scalar type has ordinality 0, the next has ordinality 1, and so on for each value in that scalar type. The ordinality of an `integer` or `longint` value is the value itself; for example, the ordinality of –10 is –10. In any scalar type, each value except the first has a predecessor based on this ordering, and each value except the last has a successor based on this ordering.
- The standard function `Ord`, described in Chapter 11, can be applied to any value of scalar type; it returns the ordinality of the value.
- The standard function `Pred`, described in Chapter 11, can be applied to any value of scalar type; it returns the predecessor of the value. For the first value in the scalar type, the result is unspecified.
- The standard function `Succ`, described in Chapter 11, can be applied to any value of scalar type; it returns the successor of the value. For the last value in the scalar type, the result is unspecified.

Scalar types are written as follows:



MPW Pascal has four predefined scalar types—`integer`, `longint`, `boolean`, and `char`—and two classes of user-defined scalar types: enumerated types and subrange types. These are described in the following sections.

The integer type

Values of type `integer` are a subset of the whole numbers. As constants, these values can be denoted as described under "Numbers" in Chapter 2. The predefined `integer` constant `maxint` is defined to be 32767. The range of the type `integer` is the set of values

```
-(maxint+1), -maxint, ... -1, 0, 1, ... maxint-1, maxint
```

that is, –32768 to +32767. These are 16-bit, 2's-complement integers.

The longint type

Values of type `longint` are a subset of the whole numbers. As constants, these values can be denoted as described under "Numbers" in Chapter 2. The predefined `longint` constant `maxlongint` is defined to be +2147483647. The range of the type `longint` is the set of values

```
-(maxlongint+1), -maxlongint, ...-1, 0, 1, ...maxlongint-1, maxlongint
```

that is, -2^{31} to $2^{31}-1$, or –2147483648 to +2147483647. These are 32-bit, 2's-complement integers. Arithmetic on `integer` and `longint` operands is done in both 16-bit and 32-bit precision, as follows:

- All `integer` constants in the range of type `integer` are considered to be of type `integer`. All `integer` constants in the range of type `longint`, but not in the range of type `integer`, are considered to be of type `longint`.
- When both operands of an operator (or the single operand of a unary operator) are of type `integer`, 16-bit operations are always performed and the result is of type `integer` (truncated to 16 bits if necessary). Similarly, if both operands are of type `longint`, 32-bit operations are always performed and the result is of type `longint`.
- When one operand is of type `longint` and the other is of type `integer`, the `integer` operand is converted to `longint`, 32-bit operations are performed, and the result is of type `longint`. However, if this value is assigned to a variable of type `integer`, it is truncated (see next rule).
- The expression on the right of an assignment statement is evaluated independently of the size of the variable on the left. For example, if variable `longVar` is declared as type `longint`, the statement `longVar := maxint+maxint` will still cause `integer` overflow. If necessary, the result of the expression is truncated or extended to match the size of the variable on the left.

An important point to remember is that each operator is applied only to its two operands, so, at most, one of those operands is converted. If the expression contains other operands, those are not necessarily converted.

For example, in the expression

```
oneInt+twoInt+threeInt+oneLongint
```

the value of `oneInt` is added to `twoInt` in a 16-bit operation, the result is added to `threeInt` in another 16-bit operation, and the result of that is converted to a `longint` value and added to `oneLongint`. The result of the expression is a `longint`.

The `Ord4` function described in Chapter 11 can be used to convert an `integer` value to a `longint` value.

◆ *Note*: Operations other than division and multiplication on `longint` values take approximately one and a half times as long as corresponding operations on `integer` values. Division and multiplication take more than twice as long.

The boolean type

The values of the `boolean` type are truth values denoted by the predefined constant identifiers `false` and `true`. These values are ordered so that `false` is "less than" `true`. The function call `Ord(false)` returns zero, and `Ord(true)` returns one.

All `boolean` variables are one byte (except in packed arrays and records). Because of this, a "garbage" byte may be allocated due to alignment of a subsequent variable (for example, a `boolean` variable followed by a `longint` or `integer` variable).

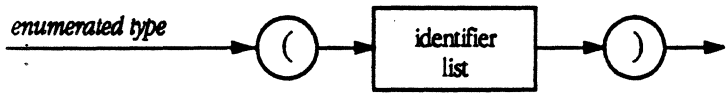
The char type

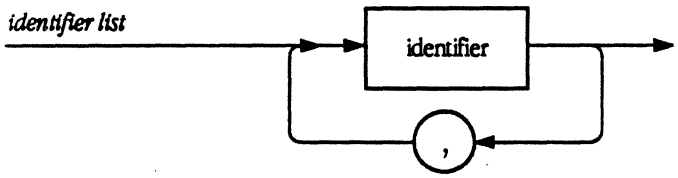
A variable of type `char` holds extended eight-bit ASCII values, represented by numeric codes in the range 0..255. The ordering of the `char` values is defined by the ordering of these numeric codes. The function call `Ord(c)`, where *c* is a `char` value, returns the numeric code of *c*. The Macintosh character set is given in Appendix C.

A `char` variable occupies two bytes of storage, except in packed arrays and records.

Enumerated types

An enumerated type defines an ordered set of values by listing the identifiers that denote these values. The ordering of these values is determined by the sequence in which the identifiers are listed.





The occurrence of an identifier within the identifier list of an enumerated type declares it as a constant for the block in which the enumerated type is declared. The type of this constant is the enumerated type being declared. These values are constants of the enumerated type in the same way that the characters `'A'`, `'B'`, and `'C'` are constants of type `char` and the integers 1, 2, and 3 are constants of type `integer`.

These are examples of enumerated types:

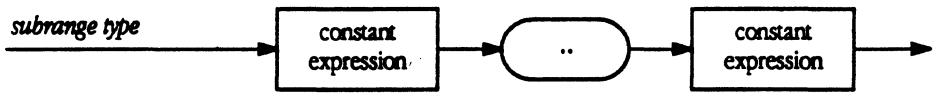
```
color = (red, yellow, green, blue)
suit = (club, diamond, heart, spade)
maritalStatus = (married, divorced, widowed, single)
```

Given these declarations, `yellow` is a constant of type `color`, `diamond` is a constant of type `suit`, and so forth. When the `Ord` function is applied to a value of an enumerated type, it returns an `integer` representing the ordering of the value with respect to the other values of the enumerated type. For example, given the declarations above, `ord(red)` returns zero, `ord(yellow)` returns one, and `ord(blue)` returns three.

- *Note*: Certain special scope rules apply to enumerated scalar types. They are described in Appendix B.

Subrange types

You define a subrange type by giving a range of values from some scalar type, called the **associated scalar type.** A subrange type provides for range checking of values within the associated scalar type. The syntax for a subrange type is



Both constants must be of scalar type. The first constant expression in a subrange type declaration must be smaller than the second constant expression. Both must be of the same scalar type, or one must be of type `integer` and the other of type `longint`. If one is of type `integer` and the other of type `longint`, the associated scalar type is `longint`.

- *Note*: When using a constant expression in a type declaration that is declaring a subrange type, you cannot use a parenthesis as the first character after the equal sign. The Compiler distinguishes subrange types from enumerated types by the first symbol after the equal sign: a left parenthesis in that position signifies an enumerated type. If a subrange specification needs parentheses, precede it with `0+`. This rule applies only within the type declaration part of a program.

Here are some examples of subrange types:

```
1..100
-10..+10
red..green
0+(const1-const2) DIV 2..const2
```

A variable of a subrange type possesses all the properties of variables of the associated scalar type, with the restriction that its runtime value must be in the specified interval. In addition, the variable may have less space allocated only if the range checking is on.

String types

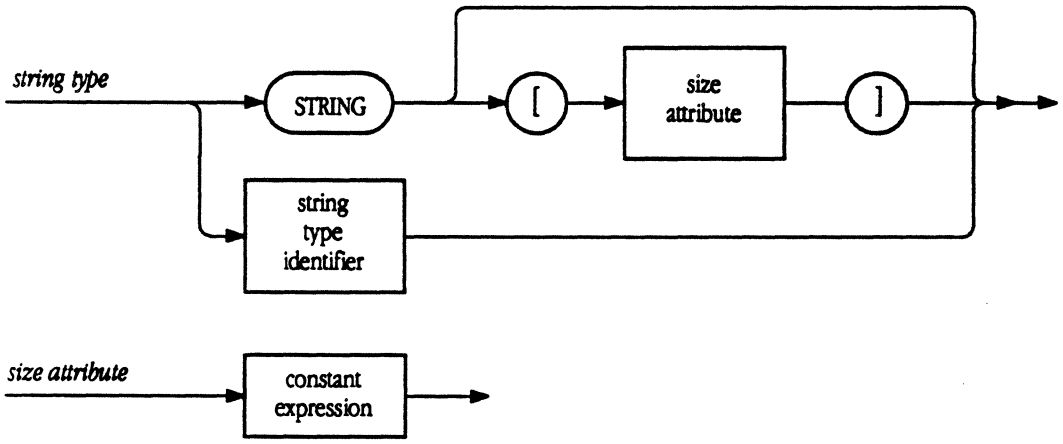
A string value is a sequence of characters that has a dynamic length attribute.

The length attribute of a string is the actual number of characters in the sequence at any time during program execution. An example of a string type declaration is

```
aString = STRING[15]
```

where `15` is the maximum size of the string. The size is the maximum limit on the length of any value of this type. The size attribute of a string type is determined when the string type is defined, and cannot change. It has a value in the range 1..255. A string type declared without a size attribute is treated as `STRING[255]`.

The length is the actual number of characters in the sequence at any time during program execution. The current value of the length attribute is returned by the standard function `Length`.



The ordering relationship between any two string values is determined by the ordering relationship of character values in corresponding positions in the two strings. The exact algorithm is given under "Comparing Strings" in Chapter 6. A capital letter does not have the same ordering value as the corresponding lowercase letter; for example, *A* is valued lower than *a*.

Remember that the size of a string is the value of the size attribute assigned to the string type when it is declared, and the length of a string is the number of characters it holds at any point, regardless of its size attribute. A program can measure the actual length of a string by using the `Length` function described under "String Procedures and Functions" in Chapter 11.

- *Note*: With a string constant, the size attribute is equal to the length—that is, the number of characters actually in the string.

Although string types are simple types by definition, they have some characteristics of structured types. As explained under "Array Types" later in this chapter, individual characters in a string can be accessed as if they were components of an array. In addition, all string types are implicitly packed types and all restrictions on packed types apply to strings. A list of these restrictions is given later in the section "Structured Types."

A string is stored as a one-byte-length field followed by the characters in the string. You can therefore change the length of the string by changing its zeroth character. For example,

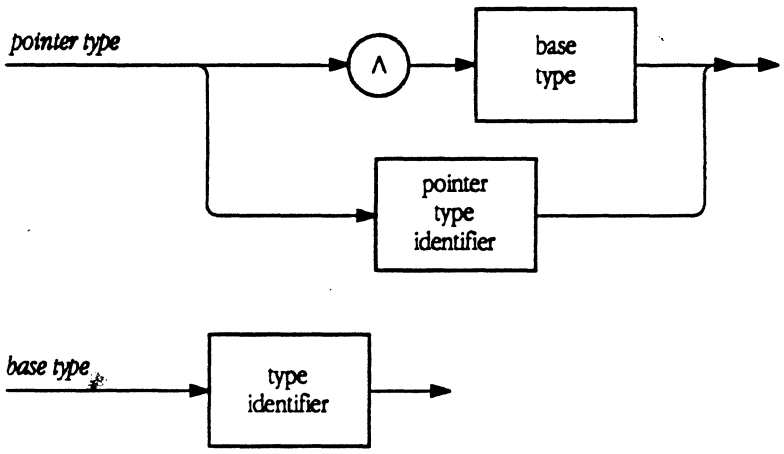
```
myStr[0] := chr(ord(myStr[0])+n);
```

changes the length of `myStr` by the value of `n`.

Operators applicable to strings are discussed in Chapter 6. Predeclared procedures and functions for manipulating strings are described in Chapter 11.

The pointer type

You can use the pointer type to define a pointer variable—a variable that holds a memory address. When you declare a pointer variable, you must specify the data type of the memory area it points to, which is then called the **base type** of that pointer variable.



The base type may be an identifier that has not yet been declared. In this case, it must be declared somewhere in the same type declaration part as the pointer type.

◆ *Note:* Certain special scope rules apply to pointer base types. They are described in Appendix B.

Aside from an address, any pointer variable can also hold the value NIL.

Conceptually, NIL is a pointer type value that does not point to anything. You can assign NIL to any pointer variable, regardless of type. However, you cannot assign the value of a pointer variable of one type to a pointer variable of another type, even if the first pointer variable has the value NIL. You assign the value NIL to a pointer variable, rather than leaving it with an undefined value, primarily because you can test for NIL.

You can create a pointer in three ways:

- By using the New procedure described in Chapter 11. This allocates a new memory area in the application heap for a **dynamic variable** and points the pointer variable to it. The size of the area is determined by the base type of the pointer variable, including optionally specified variants; see the discussion of New in Chapter 11. A dynamic variable is a variable that has no identifier of its own; the only way to access one is through a pointer.
- By using the @ operator described in Chapter 6. This points to the memory area occupied by any existing variable. The @ operator pointer function creates a pointer that is compatible with all other pointer types.
- By using the Pointer function described in Chapter 11. This allows any pointer to be coerced to any other pointer type.

Every memory address is numeric. You can use the predefined functions Ord and Ord4 to convert any address to its corresponding longint type value.

The Pointer function and the @ operator avoid the Compiler's type-checking safeguards and should be used with caution.

Chapter 5 discusses the syntax for accessing a variable pointed to by a pointer variable.

The following is an example showing how the Pointer function, the @ operator, and the New procedure can be used to access memory dynamically. Suppose you have these declarations:

```
TYPE ptr = ^longint;
   charPtr = ^char;
VAR p: ptr;
   thisLong: longint;
   cp: charPtr;
   thisStr: STRING;
```

If the address of a `longint` variable is already known, you can use the `Pointer` function to intialize the `longint` pointer `p` to it:

```
p := Pointer($904);  {Point p to address $904 in low memory.}
```

If the `longint` variable is already identified, you can use the `@` operator to point `p` to it:

```
p := @thisLong;     {Point p to memory location of thisLong.}
```

Here's another example to shows the difference: between `Pointer` and `@`:

```
p:= @cp; {p points to the pointer cp}
```

whereas

```
p:= Pointer(cp); {p and cp point to the same address}
```

- *Note*: The value of the pointer `p` remains valid only within the scope of the variable `thisLong`.

If you want to create a new memory area to hold a dynamic variable of `longint` type, you use the `New` procedure:

```
New(p);          {Point p to new heap area of longint size.}
```

Once `p` is given a value by one of the foregoing techniques, you can alter its value by the same means. For example, the following assignment moves the area pointed to by `p` four bytes toward higher memory:

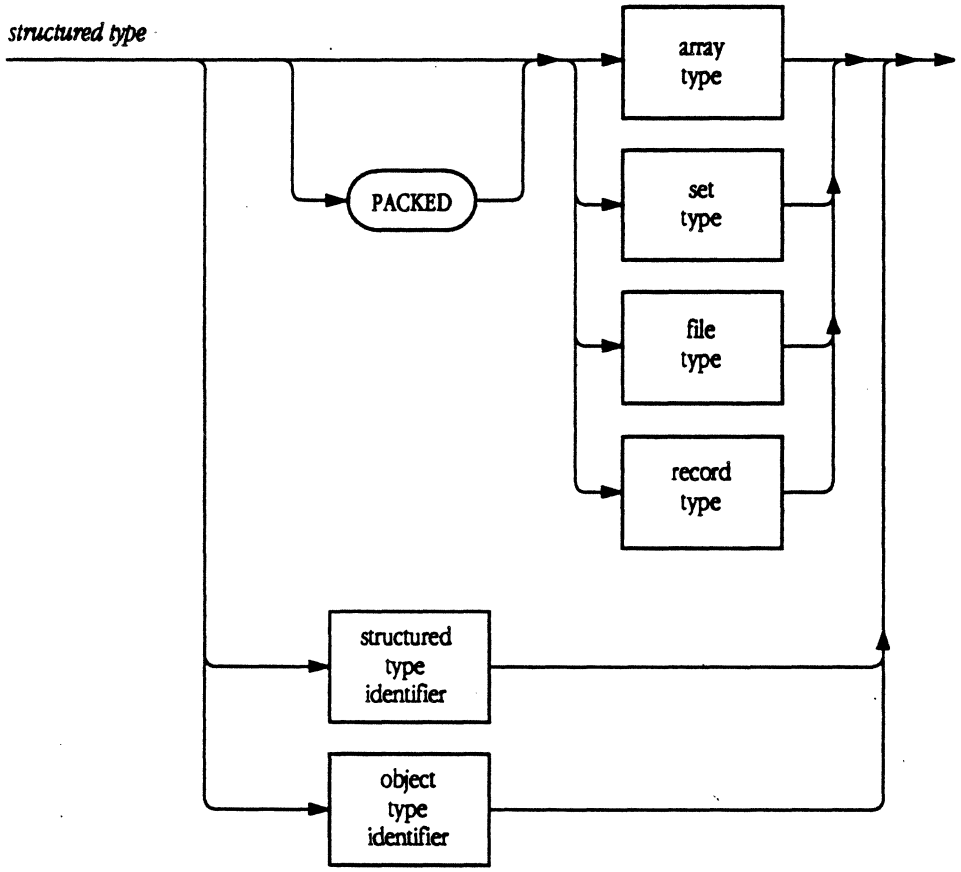
```
p := Pointer(Ord4(p)+4);        {Move pointer 4 bytes.}
```

As an example, this technique lets you access the first character in a string `thisStr` and assign its value to the `char` variable `cp`:

```
cp := Pointer(Ord4(@thisStr)+1);  {Access first char in string.}
                                  {length byte is at Ord4(@thisStr)+0}
```

Structured types

A **structured type** is a data type that stores more than one value. Each structured type is characterized by its structuring method and by the type or types of its components. If the component type is itself structured, the resulting structured type exhibits more than one level of structuring. There is no specified limit on the number of levels of structuring a data type can have.



The use of the word PACKED in the declaration of a structured type indicates to the Compiler that data storage should be economized, even if this causes less efficient access to a component of a variable of this type. Although you can use the word PACKED when declaring any structured type, PACKED only affects the storage of record and array types.

The word PACKED only affects the representation of one level of the structured type in which it occurs. If a component is itself structured, the component's representation is packed only if the word PACKED also occurs in the declaration of its type.

The @ operator is valid on byte-aligned fields of packed structures.

△ **Important** If 68000 programmers get an odd address and try to access more than a byte, they'll get an illegal address. △

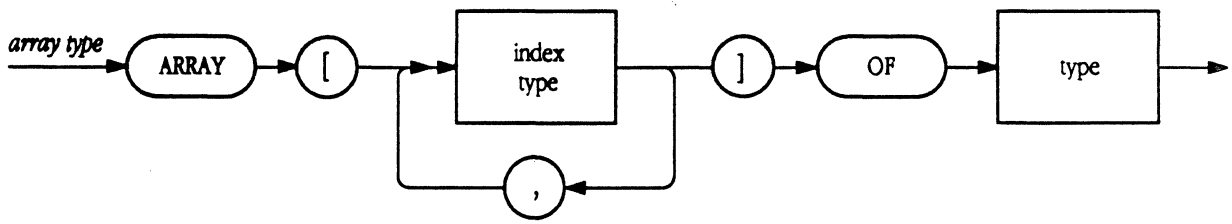
There are two restrictions on using components of packed variables:

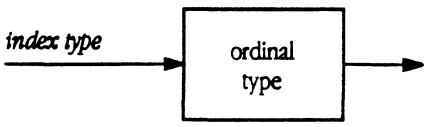
- You can only use components of variables of packed types as actual variable parameters with procedures or functions if the component is allocated on a byte boundary.
- You can only use the @ operator on components of variables of packed types if the component is allocated on a byte boundary.

The implementation of packing is complex; details of memory allocation to components of a packed variable are not specified in this manual.

Array types

An array type consists of a fixed number of components that are all of one type, called the **component type.** The number of elements is determined by one or more **index types,** one for each dimension of the array. There is no specified limit on the number of dimensions. In each dimension, the array can be indexed by every possible value of the corresponding index type, so the number of elements is the product of the number of values in each of the index types. However, static global arrays should not contain more than 32767 bytes unless the `-m` option is used. See Chapter 13 for details on the `-m` Compiler option.





The type following the word `OF` is the component type of the array and can be an existing type identifier or a new type.

◆ *Note*: The index type cannot be `longint` or a subrange of `longint`.

Here are some examples of array types:

```
ARRAY[1..100] OF real
ARRAY[boolean] OF color
ARRAY[1..Pagesize-1] OF char
```

If the component type of an array type is also an array type, the result can be regarded either as an array of arrays or as a single multidimensional array. For example,

```
ARRAY[boolean] OF ARRAY[1..10] OF ARRAY[size] OF real
```

is equivalent to

```
ARRAY[boolean, 1..10, size] OF real
```

Likewise,

```
PACKED ARRAY[1..10] OF PACKED ARRAY[1..8] OF boolean
```

is equivalent to

```
PACKED ARRAY[1..10,1..8] OF boolean
```

"Equivalent" means that the Compiler performs the same actions with the two constructions.

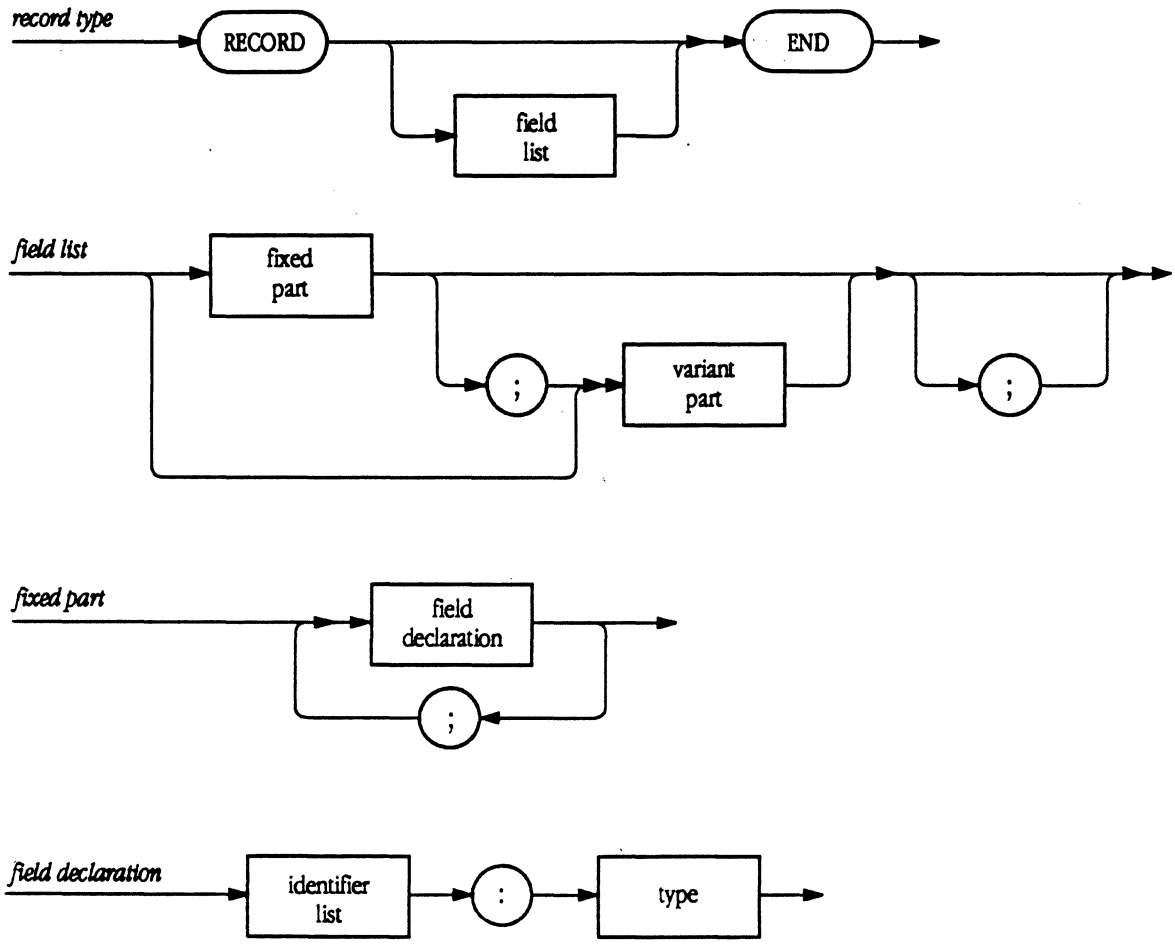
A component of an array can be accessed by following the array's identifier with one or more indexes in brackets, separated by commas. For example, the two expressions

```
myArray[5,4]
myArray[5][4]
```

both access the fourth element in the fifth subarray of the array `myArray`. For further information, see "Arrays and String Indexes" in Chapter 5.

Record types

A record type consists of a fixed number of components called **fields,** which can be of different types. For each component, the record type declaration specifies the type of the field and an identifier that names the field.

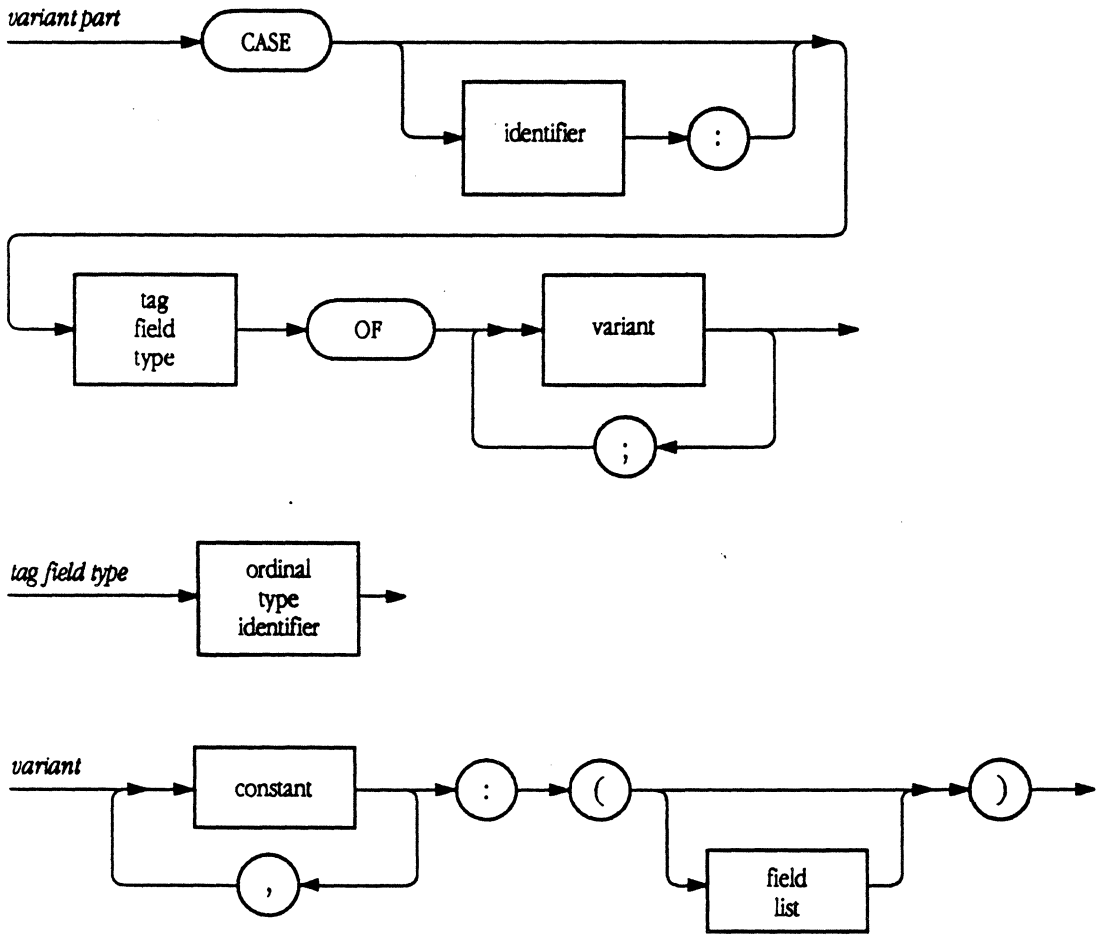


The fixed part of a record type specifies a list of "fixed" fields, giving an identifier and a type for each field. Each fixed field contains data that is always accessed in the same way.

Here is an example of a record type:

```
RECORD
 year: integer;
 month: 1..12;
 day:  1..31
END
```

A variant part allocates memory space with more than one list of fields, thus permitting the data in this space to be accessed in more than one way. Each list of fields is called a **variant.** The variants overlay each other—that is, they occupy the same space in memory.



The variant part allows for an optional identifier, called the **tag field identifier.** If a tag field identifier is present, it is automatically declared as the identifier of an additional fixed field of the record, called the **tag field.** The value of the tag field may be used by the program to indicate which variant should be used at a given time. If there is no tag field, the program must select a variant on some other criterion.

◆ *Note*: The type `longint` cannot be used as a tag type.

Each variant is identified by one or more constants. All the constants must be distinct and must be of a scalar type that is the same as or compatible with the tag type. The constants that introduce a variant are not used for referring to fields of the variant; the actual field identifiers are used. However, these constants can be used as optional arguments with the `New` procedure, described in Chapter 11.

Variant fields are accessed in exactly the same way as fixed fields.

Here are some examples of record types with variants:

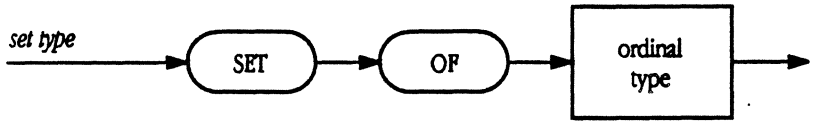
```
RECORD
 name, firstName: STRING[80];
 age: 0..99;
 CASE married: boolean OF
  true: (spousesName: STRING[80]);
  false: ()
END

RECORD
 x, y: real;
 area: real;
 CASE s: shape OF
  triangle: (side: real; inclination, angle1, angle2: angle);
  rectangle: (side1, side2: real; skew, angle3: angle);
  circle:  (diameter: real)
END
```

Set types

A set type defines a group of values, each of which has the same scalar type, called the set's *base type*. Each possible value of a set type is some subset of the possible values of the base type.

- *Note*: The base type must not have more than 2040 possible values and cannot be `longint` or `integer`. If the base type is a subrange of `integer`, all its values must be within the limits 0..2039. Because of the way sets are stored, you cannot specify a base type range such as 5000..5001.

When you create a variable of a set type, that variable can hold none, one, several, or all of the values of the set.



The set operators and the way in which set values are denoted in Pascal are discussed in Chapter 6.

Sets with fewer than 32 possible values in the base type can be held in a register and offer the quickest access time. For sets larger than that, there is a performance penalty that is essentially a linear function of the size of the base type.

The empty set `[]` is a possible value of every set type.

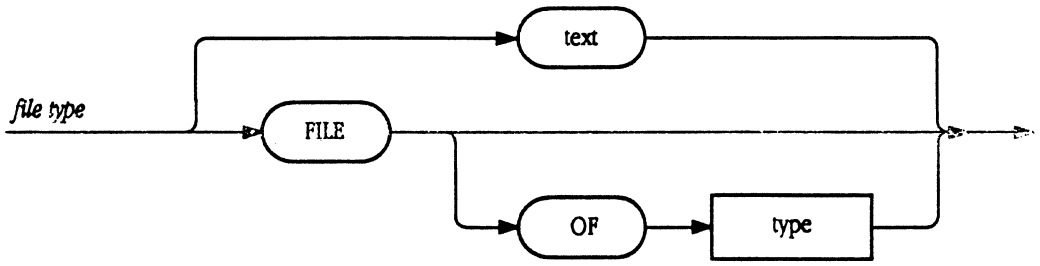
Here are some examples of set types:

```
SET OF char
SET OF (black, brown, red, yellow, white)
SET OF 1..10
names = (Eliot, Pound, Yeats) {a new scalar type}
poets = SET OF names {a set type using the new scalar type}
```

File types

A file type is a structured type consisting of a sequence of components that are all of one type, the component type. The component type may be any type except a file type or any type containing a file type.

The component data is not in program-addressable memory but is accessed by means of a peripheral device. The number of components (the length of the file) is not fixed by the file type declaration.



The type `FILE` (without the `OF TYPE` construct) represents an untyped file, for use with the `Blockread` and `Blockwrite` functions described in Chapter 10.

- *Note*: Although the symbol `FILE` can be used as a type identifier, it cannot be redeclared because it is a reserved word.

The predefined file type `text` denotes a file of characters organized into lines. The file may be stored on a file-structured device, or it may be a stream of characters from a character device such as the Macintosh keyboard. Files of type `text` are supported by the specialized I/O procedures discussed in Chapter 10.

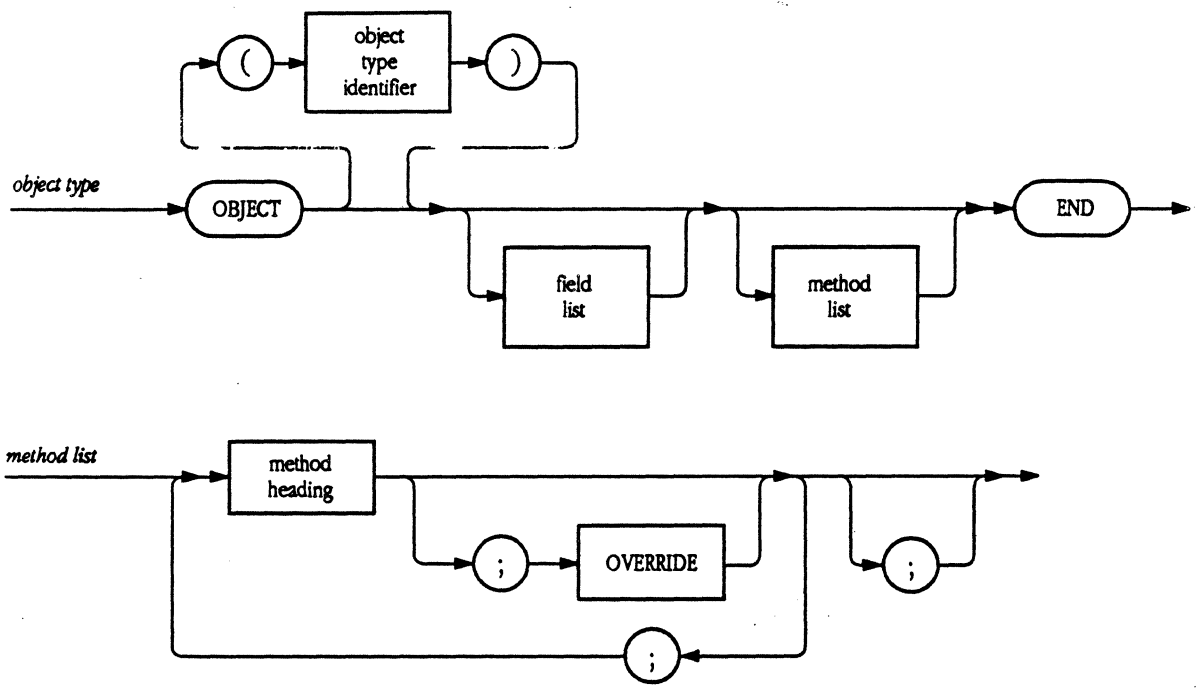
In a stored file of type `text` or `FILE OF -128..127`, the component values are packed into bytes on the storage medium. With the type `FILE OF char`, the component values of this type are stored in 16-bit words.

In MPW Pascal, files can be passed to procedures and functions as variable parameters.

Chapters 5 and 10 discuss methods of accessing file components and data.

Object types

An **object type** defines a structure for an object. An object type can have fields, like a record. The diagram for field lists above and the discussion of record type fields also apply to object type fields, except that an object type cannot have a variant part. In addition to fields, an object type can have associated procedures and functions, called *methods*.



The method heading has the syntax of a procedure or function heading, as shown in Chapter 8.

If you include the optional object type identifier and period, it must be the object type identifier that you are defining. If the method has a formal parameter list, that list must be given with the heading; similarly, if the method is a function, the type of the return value must be given with the heading.

Object types are further discussed in Chapter 12.

Type compatibility

There are three levels of type compatibility in Pascal:

- Two types may be the same. Two types are the same when they are declared using the same type identifier or when their definitions can be traced back to the same type identifier. For the rules under which user-defined anonymous types are the same, see "User-Defined Anonymous Types" at the end of this chapter.
- Two types may be compatible.
- Two types may be assignment compatible.

Compatibility and assignment compatibility are discussed below.

The same types are required only

- between actual and formal variable parameters
- between actual and formal result types of functional parameters
- between actual and formal value and variable parameters within parameter lists of procedural or functional parameters
- when a one-dimensional `PACKED ARRAY OF char` is being compared with another via a relational operator

Parameters are discussed in Chapter 8.

Assignment compatibility is usually required in other contexts, although simple compatibility is occasionally enough.

Compatible types

Compatible types are required in many contexts where two or more entities are used together, such as in expressions, in relational operations, and with `FOR` statement control variables and their initial and final values. Other specific instances where type compatibility is required are noted elsewhere in this manual.

Two types are compatible if any of the following are true:

- They are the same type.
- One is a subrange of the other.
- Both are subranges of the same type.
- Both are string types (the lengths and sizes may differ).
- Both are set types, and their base types are compatible.
- Both are of type `PACKED ARRAY OF char` and have the same number of components.

Assignment-compatible types

Assignment compatibility is required whenever a value is assigned to something, either explicitly (as in an assignment statement) or implicitly (as in passing value parameters).

A type `T2` is assignment compatible with another type `T1` in the expression `T1 := T2` if any of the following are true:

- `T1` and `T2` are identical types, and neither is a file type or a structured type that contains a file type component at any level of structuring.
- `T1` is a real type, and `T2` is type `integer`.
- `T1` and `T2` are compatible scalar types, and the value of `T2` is within the range of possible values of `T1`.
- `T1` and `T2` are compatible set types, and all the members of `T2` are within the range of possible values of the base type of `T1`.
- `T1` and `T2` are string types, and the current length of `T2` is equal to or less than the size attribute of `T1`.
- `T1` and `T2` are both type `PACKED ARRAY OF char`.
- `T1` is a string type with size greater than zero or a `char` type, and `T2` is a quoted character constant.
- `T1` is type `PACKED ARRAY[1..n] OF char`, or actually has *n* elements, and `T2` is a string constant containing exactly *n* characters. This is not true, however, if $n=1$, because a string constant of length 1 is a quoted character constant.
- `T1` is an object type, and `T2` is an object reference to the same type or a descendant type.

Whenever assignment compatibility is required and none of the above is true, either a Compiler error or a runtime error occurs.

Type coercion

A value or variable access of one type can be changed into a value of another type with the syntax

typeID(x)

Using this construction, *x* can be a variable identifier, a variable identifier plus one or more qualifiers (array index, field designator, file buffer symbol, or pointer symbol), or an expression. You can use this syntax on the left or right side of an assignment statement.

The term *typeID* stands for any type identifier. The expression *typeID* (*x*) is treated as an instance of the type specified by the term *typeID,* provided that the storage size of *x* is not changed. For conversion between scalar types, the resulting storage size can be different.

▲ **Warning** Constants cannot be type coerced to a structured type. ▲

Here are some examples of type coercion:

```
TYPE
 ARecord = RECORD
       x, y: integer
      END;
VAR
 recordVar: ARecord;
 LongVar: longint;
 IntVar: integer;
---
recordVar := ARecord(LongVar);
longint(recordVar) := 34 + 65536*180;
IntVar := integer(LongVar);
```

The last line shows a conversion from a four-byte quantity to a two-byte quantity, which is allowed for scalar types. In this case, the conversion is checked for overflow if overflow checking is in effect. (Overflow checking can be controlled by the `$ov` Compiler option described in Chapter 13.)

▲ **Warning** Using type coercion to widen a variable can alter adjacent memory locations, for example, `longint(anInteger) := 5;`. ▲

You can also use type coercion for object types. In that case, the coerced value must be a member of the type into which it is coerced. The coercion is checked for legality only if `$R` range checking is in effect.

MPW Pascal does not support type coercion of set variables.

Type declarations

Any program, procedure, or function that declares type identifiers contains a type declaration part, as discussed in Chapter 3.

Here is an example of a type declaration part:

```
TYPE count = integer;
   range = integer;
   color = (red, yellow, green, blue);
   sex   = (male, female);
   year  = 1900..1999;
   shape = (triangle, rectangle, circle);
   card  = ARRAY[1..80] OF char;
   str   = STRING[80];
   polar = RECORD r: real; theta: angle END;
   person = ^personDetails;
   personDetails = RECORD
    name, firstName: str;
    age: integer;
    married: boolean;
    father, child, sibling: person;
    CASE s: sex OF
     male: (enlisted, bearded: boolean);
     female: (pregnant: boolean)
    END;
   people = FILE OF personDetails;
   intfile = FILE OF integer;
```

In the example above, `count`, `range`, and `integer` denote identical types. The type `year` is compatible with, but not identical to, the types `range`, `count`, and `integer`.

User-defined anonymous types

You can give a type to a variable without defining a type identifier. In that case, the type declaration appears on the right side of a variable declaration, in place of a type identifier. This is called a **user-defined anonymous type.** You can create the same structures with these types that you can with identified types, except that you cannot use them in the following contexts:

- formal parameter lists
- definitions of functional return values
- definitions of object reference variables

Here are some examples of user-defined anonymous type declarations, appearing on the right sides of variable declarations:

```
VAR  var1: (red, yellow, green, blue);
   var2: STRING[80];
   var3: RECORD
       name, firstName: str;
       age: integer;
       married: boolean;
       father, child, sibling: person;
       CASE s: sex OF
        male: (enlisted, bearded: boolean);
        female: (pregnant: boolean)
       END;
       END;
```

User-defined anonymous types that are separately declared are never the same types. For example, the declarations

```
var1: RECORD
    a, b: real
   END;
var2: RECORD
    a, b: real
   END;
```

do not give `var1` and `var2` the same type, even though they appear to be exactly the same. However, two variables declared in the same user-defined anonymous type declaration, as in

```
var1, var2: RECORD
       a, b: real
      END;
```

are of the same type.

You can coerce two separately declared user-defined anonymous types into being treated as identical by using the type coercion technique described earlier in this chapter.

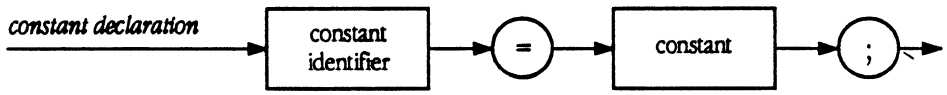
Chapter 5 **Constants and Variables**

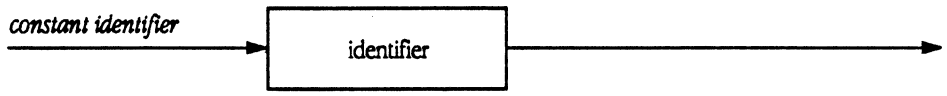
THE DATA PROCESSED BY A PASCAL PROGRAM IS HELD in **constants** and **variables**, which must be declared before they can be used. For a discussion of how these declarations fit into programs and units, see Chapter 9. ■

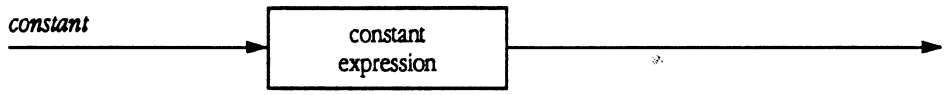
Contents

Constant declarations

A constant declaration defines an identifier that denotes a constant within the block that contains the declaration. The scope of a constant identifier does not include its own declaration. See Chapter 3 for a discussion of scope.

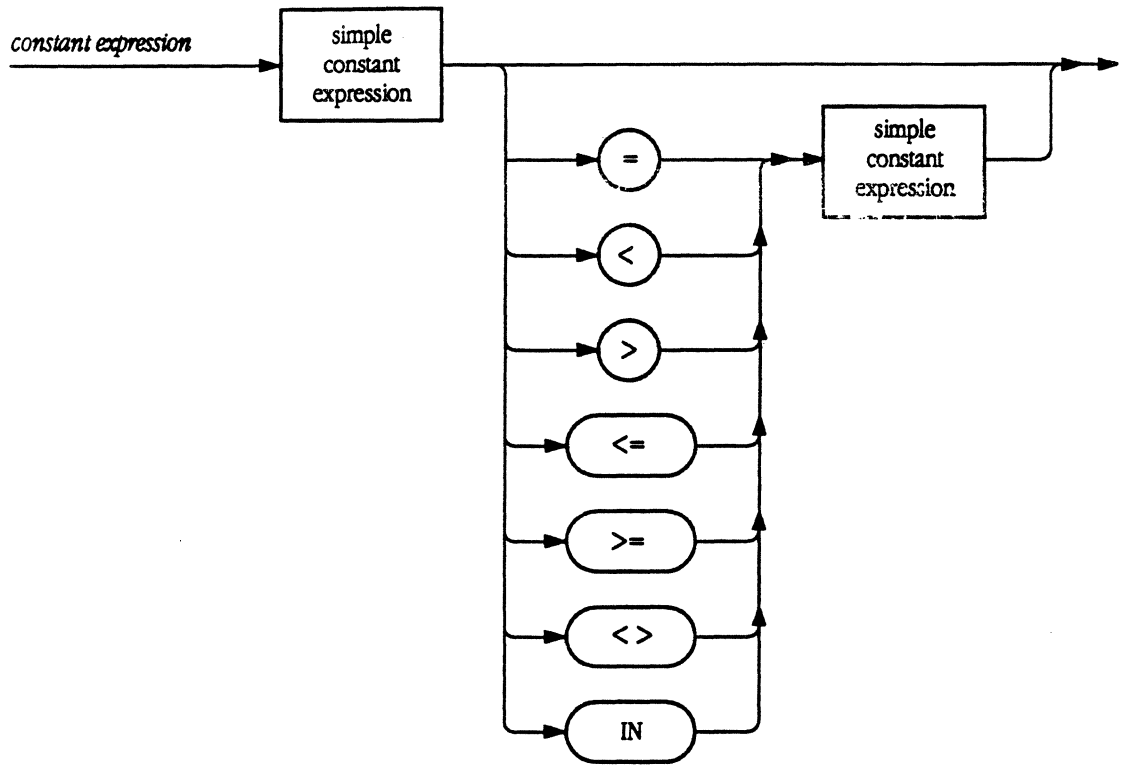


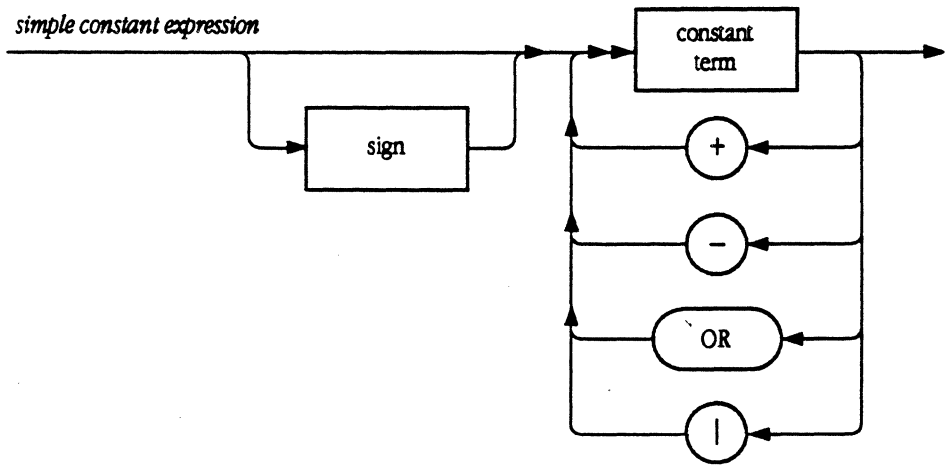




Constant expressions

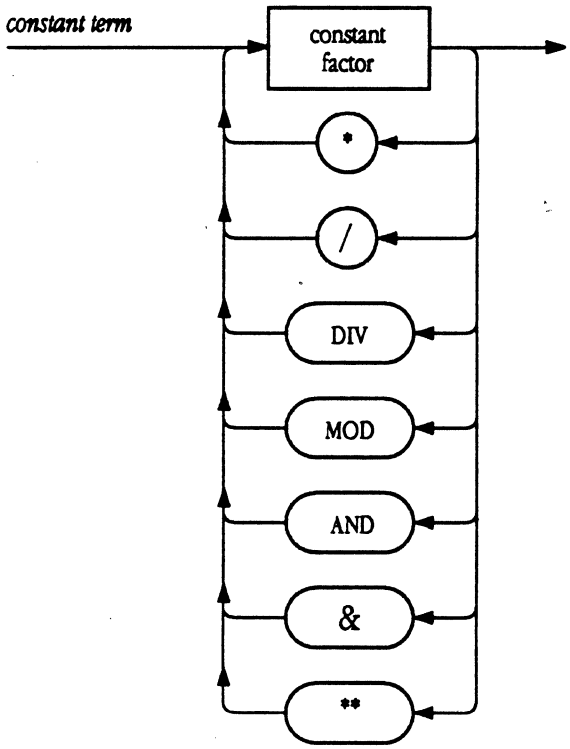
A constant expression is an expression that denotes a constant. You can write a sign in front of a constant expression whenever that expression's value is a number. For example, you can use a minus sign with the predefined constant `maxlongint` to denote the value -2147483647.

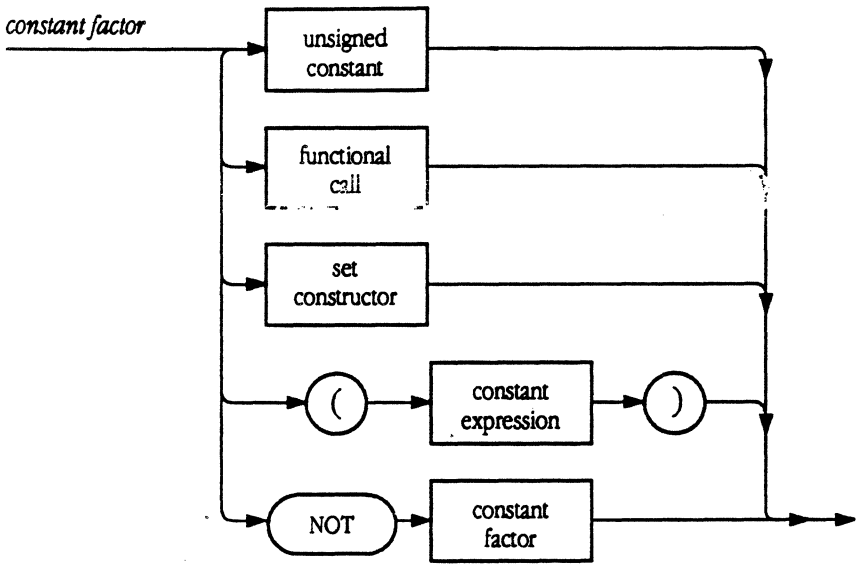




Constant expressions can be used wherever a single constant is allowed. This means that constant expressions are allowed in the constant declarations described in the preceding section, in subranges, and as CASE constants. (Subranges are described in Chapter 4, and CASE statements in Chapter 7.) Constant expressions are evaluated by the Compiler.

Constant expressions follow the same rules as other expressions. Operands must be compatible with their operators (+, -, *, **, DIV, /, IN, AND, OR, NOT, &, |, and relational operators). Constant sets may be defined within the CONST section, and set operations are permitted. You can use an index to refer to a single character in a string constant, as described later in the section "Arrays and String Indexes."





These functions are permitted in constant expressions:

```
Abs Sqr Odd Ord Ord4 Chr Trunc Round Sizeof
```

When using `Sizeof` in constant expressions, only a single type or variable identifier is allowed. (The `Sizeof` function ordinarily allows field specifications.)

All integer arithmetic is performed using `longint` values (32-bit integers). All real arithmetic is performed using `extended` (80-bit) values.

Here is an example of the use of constant expressions:

```
TYPE
 Color    = (Blue, Cyan, Green, Yellow, Red, Magenta);
CONST
 PageSize  = 1024;
 NbrOfBlks = PageSize DIV 512;
 WhiteColor = [Blue, Green, Red];
 BlackColor = [Cyan, Yellow, Magenta];
VAR
 InputBufr: PACKED ARRAY [0..PageSize - 1] OF char;
 AColor: Color;
BEGIN
 ---
 Read(Input, Ch);
 AColor := AssignColor(Ch);
 IF Acolor IN WhiteColor THEN ShowLight;
 IF Acolor IN BlackColor THEN ShowDark;
 ---
END.
```

◆ *Note:* When using a constant expression in a type declaration that is declaring a subrange type, you cannot use a parenthesis as the first character after the equal sign. The Compiler distinguishes subrange types from enumerated types by the first symbol after the equal sign: a left parenthesis in that position signifies an enumerated type. If a subrange specification needs parentheses, precede it with "0+".
Here are two examples:

```
TYPE
 range = 0+(const1-const2) DIV 2..const2;
 color = (black, brown, red, orange, yellow, green);
```

This rule only applies within the type declaration part of a program.

Predefined numeric constants

The predefined constant `maxint` is of type `integer`. Its value is 32767. This value satisfies the following conditions:

- Any unary operation performed on a whole number in this interval is correctly performed according to the mathematical rules for whole-number arithmetic, with the exception of `-(-maxint-1)`.
- Any binary integer operation on two whole numbers in this same interval is correctly performed according to the mathematical rules for whole-number arithmetic, provided that the result is also in this interval. If the mathematical result is not in this interval, then the actual result is the low-order 16 bits of the mathematical result. Note that the sign of the actual result will sometimes be the opposite of the mathematical result in this case.
- Any relational operation on two whole numbers in this same interval is correctly performed according to the mathematical rules for whole-number arithmetic.

◆ *Note:* Two operations do not work correctly, even though they are technically in the correct interval. They are

```
-(-maxint-1)
abs(-maxint-1)
```

The predefined constant `pi` is the representation, in `extended` format, of the value of π. This value is precise to 19 decimal digits: 3.141592653589793239.

The predefined constant `inf` represents positive infinity in an extended format.

The predefined constant `maxcomp` is the maximum `comp` value, 9223372036854775807.

The following predefined constants are the smallest normalized values for each data type: `minnormreal` has the value 2^{-126}; `minnormdouble` has the value 2^{-1022}, and `minnormextended` the value 2^{-16383}.

The predefined constant `compsecs` is of type `longint`. Its value holds the compilation date/time in seconds, as described in the "OSUtils" section of *Inside Macintosh*.

Predefined string constants

MPW Pascal includes two predefined string constants that are evaluated at compile time. They are intended for version control. The constant `compdate` holds the compilation date, and `comptime` holds the compilation time.

The two constants act as if you specified them in the constant declaration part of your program as

```
compdate = 'MM/DD/YY';
```

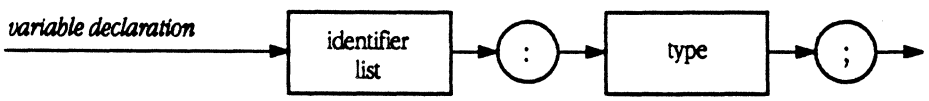
where MM is the month, DD is the day of the month, YY is the year; and

```
comptime = 'HH:MM:ss AM/PM';
```

where HH is the hour (in 24-hour format), MM the minute, ss the second, AM morning, and PM afternoon.

Variable declarations

A variable declaration consists of a list of identifiers denoting new variables, followed by their type.



The syntax for an identifier list is given under "Enumerated Types" in Chapter 4.

The type given for a variable can be a type identifier declared in a preceding type declaration part (which can be in the same block or an enclosing block, or in a unit) or a new type definition (a user-defined anonymous type). User-defined anonymous types are discussed at the end of Chapter 4.

The occurrence of an identifier within the identifier list of a variable declaration declares it as a variable identifier for the block in which the declaration occurs. The variable can then be referred to throughout the remainder of that block, unless the identifier is redeclared in an enclosed block as described in Chapter 3. If it is redeclared, the redeclaration creates a new variable that uses the same identifier and does not affect the value of the original variable.

The values of all variables are undefined at the start of each activation of a block. The main program block is activated when the program is run. A procedure or function block is activated each time the procedure or function is called.

Local data (that is, data declared within a procedure or function) may be greater than 32K. The code generated for references to variables beyond the first 32K will be less efficient than that generated for variables within the first 32K.

Here are some examples of variable declarations. There may not be more than 32K of global data declared unless the `-m` option is used. The `-m` option will generate less efficient code. See Chapter 13 for details.

```
x, y, z: real;
i, j: integer;
k: 0..9;
p, q, r: boolean;
operator: (plus, minus, times);
a: ARRAY[0..63] OF real;
c: color;
f: FILE OF char;
huel, hue2: SET OF color;
pl, p2: person;
m, ml, m2: ARRAY[1..10, 1..10] OF real;
coord: polar;
pooltape: ARRAY[1..4] OF tape;
```

▲ **Warning** The 32K of globals limit is inherent in the current run-time architecture (without the `-m` option), and applies to all object files at link time. Because this limit includes the implementation globals of units and even the object files from other languages, the Pascal compiler cannot detect a problem in all cases. The Linker will give an error, however. ▲

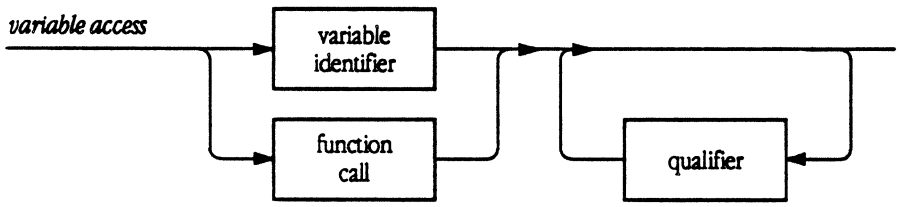
Variable accesses

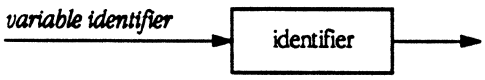
When a variable's identifier is used in a program, it is called an **access** of that variable. You use a variable access to do any of the following:

- obtain the value of a variable
- assign a value to a variable
- pass the value of a variable to a procedure or function

The object accessed may be any one or a combination of the following:

- a simple variable
- a pointer variable
- the collection represented by a variable of structured type
- a part of a structured type
- the identified variable of a pointer
- the identified object of an object reference variable
- a variable reached through a function call

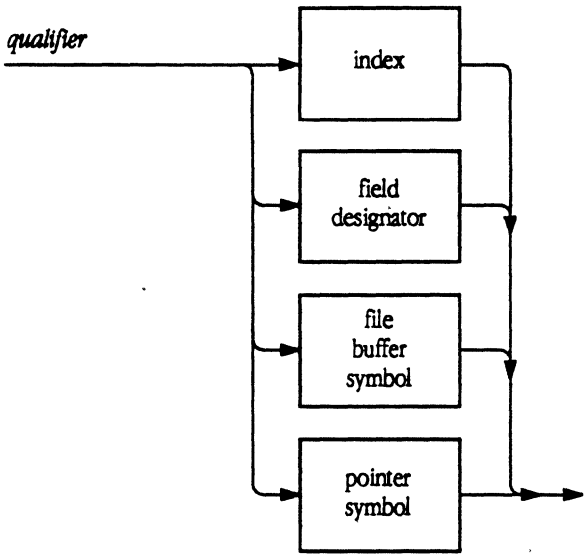




Syntax for the various kinds of qualifiers used with variable accesses is given below.

Qualifiers

In a variable access, the variable identifier or function call can be followed by one or more **qualifiers.** Each qualifier modifies the meaning of the variable access.



As indicated in the diagram for variable accesses, there can be none, one, or more than one qualifier following a variable identifier or function call, depending on the levels of structure in the variable and on which particular level you want to access.

For example, an array identifier with no qualifier is a reference to the entire array:

```
xResults
```

If the array identifier is followed by an index, this denotes a specific component of the array, which may be a simple or structured variable:

```
xResults[current+1]
```

If the array component is a record or object reference, the index may be followed by a field designator; in this case, the variable access denotes a specific field within a specific array component:

```
xResults[current+1].link
```

If the field is a pointer, the field designator may be followed by the pointer symbol to denote the variable pointed to by the pointer:

```
xResults[current+1].link^
```

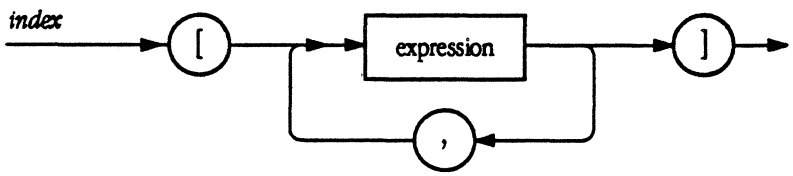
If the variable identified by the pointer is an array, another index can be added to denote a component of this array, and so forth:

```
xResults[current+1].link^[i]
```

Arrays and string indexes

A specific component of an array variable is denoted by a variable access that refers to the array variable, followed by an index that specifies the component.

A specific character within a string variable or constant is denoted by a variable access, quoted string constant, or string constant identifier, followed by an index that specifies the character position.



The index of a string always ranges from 0 to a maximum of 255, depending on the size and length of the string. The index of an array depends on the ordinal type or types defined as the array's index type or types.

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index types in the array declaration, and the type of each expression must be assignment compatible with the corresponding index type.

In indexing a multidimensional array, you can use either multiple indexes or multiple expressions within an index. The two forms are equivalent. For example,

```
m[i][j]
```

is equivalent to

```
m[i,j]
```

A string value can be indexed by only one index expression, whose value must be in the range 1..*n*, where *n* is the current length of the string value. The effect is to access one character of the string value.

♦ *Note:* In general, you cannot assign a value to an individual character position in a string unless a character previously occupied that position.

When a string value is manipulated by assigning values to individual character positions, the dynamic length of the string is not maintained. For example, suppose that `strval` is declared as follows:

```
strval: string[10];
```

Pascal allocates one byte for a number that represents the current length of the string, followed by space for ten `char` values. (The dynamic length is the number of `char` values in the string at any given time.) Initially, all of this space contains unspecified values. The assignment

```
strval[1] := 'F'
```

may or may not work, depending on what the unspecified length happens to be. If this assignment works, it stores the `char` value `F` in character position 1, but the length of `strval` remains unspecified. Therefore, the effect of a statement such as `Writeln(strval)` is unspecified.

You do not have to worry about this if you are dealing with the entire string. The statement

```
strval := 'F'
```

always works. The dynamic length of the string acquires a value of one, and the statement `Writeln(strval)` prints an `F`. The values of character positions beyond position 1 are still unspecified, and the subsequent assignment

```
strval[2] := 'F'
```

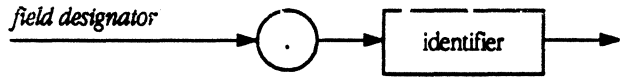
leaves `strval` unchanged, because you cannot change the length of the string by assigning a string element as a character.

The predefined procedures for string manipulation, described in Chapter 11, always properly maintain the lengths of the string values they modify, and so are easier to use than this kind of indexed string manipulation.

The zeroth position of a string is special: it contains the dynamic length of the string. For details, see "String Types" in Chapter 4.

Records and field designators

A specific field of a record variable is denoted by a variable access that refers to the record variable, followed by a field designator that specifies the field.



Here are some examples of field designators:

```
employee.salary
coord.theta
```

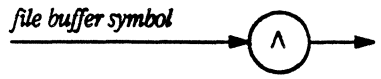
The period (.), as well as the record variable identifier or function call, can be omitted inside a `WITH` statement that lists the record variable identifier or function call. See Chapter 7 for more information about the `WITH` statement.

File window variables

Although a file variable may have any number of components, only one component is accessible at any time. The position of the current component in the file is called the **current file position.** See Chapter 10 for predefined procedures that move the current file position. Program access to the current component uses a special variable associated with the file, called a **file window variable.**

The file window variable is implicitly declared when the file variable is declared. If `F` is a file variable with components of type `T`, the associated file buffer is a variable of type `T`.

The file window variable associated with a file variable is denoted by a variable access that refers to the file variable, followed by a qualifier called the file buffer symbol.



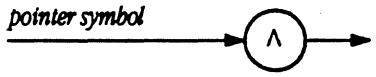
Thus, the file window variable of file `F` is referred to by `F^`.

Chapter 10 describes the predefined procedures that are used to move the current file position within the file and to transfer data between the file window variable and the current file component.

Pointers and their identified variables

The value of a variable is either NIL or a value that identifies some other variable, called the **identified variable** of the pointer. Pointer types are discussed in Chapter 4.

The identified variable of a pointer variable is accessed by using the pointer variable followed by a pointer symbol qualifier.



The constant NIL does not point to a variable. If you access memory by means of a NIL pointer reference, the results are unspecified. However, there may not be any error indication.

Here are some examples of references using pointers:

```
p1^
p1^.sibling^
```

Object references

A variable that is declared using an object type is an object reference variable. Object reference variables are discussed in Chapter 12.

Chapter 6 **Expressions**

EXPRESSIONS CONSIST OF OPERANDS and (usually) operators. Operands consist of the following:

- constants
- variables
- function calls
- set constructors

Constants and variables are discussed in Chapter 5. Function calls and set constructors are described later in this chapter.

Operators make up a subset of the special symbols described in Chapter 2. They are described below. The rules for writing expressions are given at the end of this chapter. ■

Contents

Operators

Table 6-1 shows the MPW Pascal operators and their precedence:

■ **Table 6-1** Precedence of operators

Operation	Precedence	Category
@, NOT, **	highest	exponent and unary operators
*, /, DIV, MOD, AND, &	second	"multiplying" operators
+, −, OR, |	third	"adding" operators and signs
=, <>, <, >, <=, >=, IN	lowest	relational operators

Operations with equal precedence are performed from left to right, in general. The single exception is for the exponentiation operator (**). Multiple exponentiation operations are performed right to left.

Subexpressions that are not related by precedence may be evaluated in any order.

Arithmetic operators

The types of operands and results for arithmetic binary and unary operations are shown in Tables 6-2 and 6-3.

■ **Table 6-2** Binary arithmetic operators

Operator	Operation	Operand type	Type of result
+	addition	integer, longint, or real type	integer, longint, or extended
-	subtraction	integer, longint, or real type	integer, longint, or extended
*	multiplication	integer, longint, or real type	integer, longint, or extended
**	exponentiation	integer, longint, or real type	integer, longint, or extended
/	division	integer, longint, or real type	extended
DIV	division with integer result	integer or longint	integer or longint
MOD	remainder	integer or longint	integer

The symbols +, -, and * are also used as set operators (described later in this chapter).

◆ *Note:* Except for | and &, the Compiler may evaluate the operands of a binary operator in either order. With | and &, the left operand is evaluated first, then the right operand only if required to determine a value.

The real types are `real` (or `single`), `double`, `extended`, and `comp` (or `computational`).

Use of the exponentiation operator when either operand is a real type requires linking to the SANE library.

■ **Table 6-3** Unary arithmetic operators (signs)

Operator	Operation	Operand type	Type of result
+	identity	integer, longint, or real type	integer, longint, or extended
-	sign negation	integer, longint, or real type	integer, longint, or extended

Any operand whose type is a subrange of a scalar type is treated as if it were of the scalar type.

If both the operands of an addition, subtraction, or multiplication operator are of type `integer` or `longint`, the result is of type `integer` or `longint` as described in Chapter 3; otherwise, the result is of type `extended`.

The result of the identity or sign negation operator is of the same type as the operand, except that real types are converted to `extended`.

The value of *i* `DIV` *j* is the mathematical quotient of *i*/*j*, rounded toward zero to an `integer` or `longint` value. An error occurs if *j* = 0.

The `MOD` operator returns the remainder of the division of its two operands. That is,

```
(-i ) MOD j = -(i MOD j )
```

The value of *i* `MOD` *j* is equal to the value of

```
i-(i DIV j )*j
```

The sign of the result of `MOD` is always the same as the sign of *i*. An error occurs if the value of *j* is zero.

- *Note:* The name `MOD` is actually a misnomer; `MOD` returns the remainder after division of *i* by *j*. To obtain the modulus, a number in the closed interval from 0 to *j* – 1, use the expression

```
((i MOD j )+j ) MOD j
```

Boolean operators

The types of operands and results for Boolean operations are shown in Table 6-4.

Table 6-4 Boolean operators

Operators	Operation	Operand type	Type of result
`OR`, |	disjunction	`boolean`	`boolean`
`AND`, &	conjunction	`boolean`	`boolean`
`NOT`	negation	`boolean`	`boolean`

Notice that there are two operators each for disjunction and conjunction. When you use the AND and OR operators, the Compiler evaluates the entire expression, even if that is not necessary. For example, consider the expression

```
true OR boolTst(x)
```

where `boolTst` is a function that returns a `boolean` value. This expression will always have the value `true`, regardless of the result of `boolTst(x)`. However, `boolTst(x)` will always be called. This could be important if `boolTst` has side effects.

With the expression

```
true | boolTst(x)
```

evaluation stops as soon as the value `true` is reached, because evaluation proceeds from left to right for operators of the same precedence. The ampersand (`&`) and vertical bar (`|`) are referred to as the **short-circuit operators.**

▲ **Warning** Pascal does not support mixing short-circuit and normal operations and normal operators in the same expression. ▲

Set operators

The types of operands for set operations are shown in Table 6-5.

■ **Table 6-5** Set operators

Operator	Operation	Operand Type
+	union	compatible set types
-	difference	compatible set types
*	intersection	compatible set types

Result types in set operations

The following rules govern the type of the result of a set operation where one or both of the operands are a `SET OF subr`. In the statement of these rules, `ordtyp` represents any scalar type and `subr` represents a subrange of `ordtyp`:

- If `ordtyp` is not type integer, then the type of the result is type SET OF `ordtyp`.
- If `ordtyp` is type integer, then the type of the result is type SET OF 0..2039. This rule results from the limitations described under "Set Types" in Chapter 4.

Relational operators

The types of operands and results for relational operations are shown in Table 6-6.

Table 6-6 Relational operators

Operator	Operation	Operand type	Type of result
=	equal to	compatible set, simple, or pointer types	`boolean`
<>	not equal to	pointer types	
<	less than	compatible simple types	`boolean`
>	greater than	compatible simple types	`boolean`
<=	less than or equal to	compatible simple types	`boolean`
>=	greater than or equal to	compatible simple types	`boolean`
<=	subset of	compatible set types	`boolean`
>=	superset of	compatible set types	`boolean`
`IN`	member of	left operand: any scalar type `T`; right operand: type `SET OF T`	`boolean`

Comparing numbers

When the operands of =, <>, <, >, >=, or <= are numeric, they need not be of compatible type if one operand is of real type and the other is `integer` or `longint`.

- *Note:* Because of extensions provided for use with the Standard Apple Numeric Environment (SANE), the result of a comparison can be unordered. An unordered result occurs from a comparison involving a NaN (Not a Number). One important effect is that NOT (a < b) is true if either a is greater than b or a and b are unordered. You can use the `relation` function, which is included in the SANE library, to test for an unordered comparison. The SANE library is described in Appendix G.

See Appendix G and the *Apple Numerics Manual* for more information on relational operations with real operands.

Comparing booleans

If *p* and *q* are `boolean` operands, then *p* = *q* denotes their equivalence, *p* <> *q* denotes the logical exclusive-or operation, and *p* <= *q* denotes the logical expression "*p* implies *q*" (because `false<true`). You can also write NOT *p* OR *q* for logical implication.

Comparing strings

When the relational operators =, <>, <, >, <=, and > are used to compare strings, they order them according to the ordering of the ASCII character set. Note that any two string values can be compared because all string values are compatible. String comparisons follow these steps:

1. The two strings are compared a character at a time, starting with the first character.
2. Two corresponding characters are compared. If the ASCII value of one character is greater than the other, then the corresponding string is greater than the other.
3. If the two corresponding characters are equal, the point of comparison advances to the next character in each string, and the process returns to step 2.
4. If the end of one string has been reached, its value is less than the other string.
5. If the ends of both strings have been reached, the two strings are equal.

- *Note:* A set of utilities that apply ordinary language rules for comparing strings is included in the Macintosh ROM routines. If you use those utilities to compare strings rather than using the operators described here, alphabetization will follow the local language's rules rather than the ASCII table. These routines are documented in the International Utilities chapter in *Inside Macintosh.*

Comparing sets

If *u* and *v* are set operands, their comparisons have these results:

- *u* <= *v* is true if *u* is included in *v*.
- *u* >= *v* is true if *v* is included in *u*.
- *u* = *v* is true if *u* and *v* contain exactly the same members.
- *u* <> *v* is true if either *u* or *v* contains a member not contained in the other.

Testing set membership

The `IN` operator yields the value `true` if the value of the scalar type operand is a member of the set type operand; otherwise, it yields the value `false`.

Comparing packed arrays of char

In addition to the operand types shown in the table, the = and <> operators can also be used to compare a `PACKED ARRAY` [*m*..*m* + *n* − 1] `OF char` with a string constant containing exactly *n* characters, or to compare two one-dimensional `PACKED ARRAYS OF char` of identical type. The comparison follows the steps given above under "Comparing Strings."

The @ operator

A pointer to a variable (an address) can be computed with the `@` operator. The operand and result types are shown in Table 6-7.

- **Table 6-7** The pointer operator

Operator	Operation	Operand	Type of result
`@`	pointer formation	variable, parameter, procedure, or function	pointer

The `@` operator is a unary operator taking a single variable, parameter, procedure, or function as its operand and computing the value of its pointer. The type of the value is equivalent to the type of `NIL` and consequently can be assigned to any pointer variable. The pointer type is discussed in Chapter 4.

- *Note:* Objects and identified variables of handles are relocatable and may be moved at any time. Therefore, using @ on handles and object reference variables produces addresses that may not be useful.

The @ operator with a variable

For an ordinary variable (not a parameter), the use of @ is straightforward. For example, given the declarations

```
TYPE twochar = PACKED ARRAY[0..1] OF char;
VAR int: integer;
  twocharptr: ^twochar;
```

the statement

```
twocharptr := @int
```

causes `twocharptr` to point to `int`. Now `twocharptr^` is a reinterpretation of the bit value of `int` as though it were a `PACKED ARRAY[0..1] OF char`.

- *Note:* The @ operator is valid on byte-aligned fields of packed structures. For example:

  ```
  @mystring[3]
  ```

  is valid. (See "Structured Types" in Chapter 4 for details.)

The @ operator with a value parameter

When @ is applied to a formal value parameter, the result is a pointer to the stack location containing the actual value. Suppose that `name` is a formal value parameter in a procedure and `nameptr` is a pointer variable. The statement `nameptr := @name` gives `nameptr^` the value of `name`.

The @ operator with a variable parameter

When @ is applied to a formal variable parameter, the result is a pointer to the actual parameter (the pointer is taken from the stack). Suppose that `fum` is a formal variable parameter of a procedure, `fie` is a variable passed to the procedure as the actual parameter for `fum`, and `fumptr` is a pointer variable.

If the procedure executes the statement

```
fumptr := @fum
```

then `fumptr` is a pointer to `fie`. The pointer `fumptr^` denotes `fie` itself.

The @ operator with a procedure or function

It is possible to apply @ to a procedure or a function, yielding a pointer to its entry point. Note that Pascal provides no mechanism for using such a pointer. The only use for a procedure pointer is to pass it to an assembly-language routine, which can then jump to that address using the assembly-language JSR instruction.

◆ *Note:* Procedures or functions used with @ should be at the outermost nesting level.

If the procedure pointed to is in the local segment, @ returns the current address of the procedure's entry point. If the procedure is in some other segment, however, @ returns the address of the jump-table entry for the procedure. The generation of current-address or jump-table code can be controlled by means of the $B Compiler directive described in Chapter 13. When a Pascal routine address is passed to a Macintosh ROM routine, the jump table must be used.

If the procedure's segment is unloaded, code swapping may change a local segment procedure address without warning, and the procedure pointer can become invalid. If the procedure is not in the local segment, the jump-table-entry address will remain valid despite swapping because the jump table is not moved.

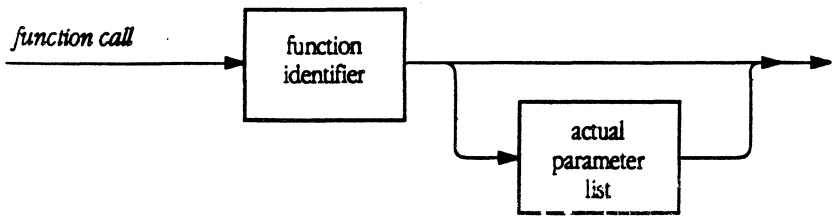
Function calls

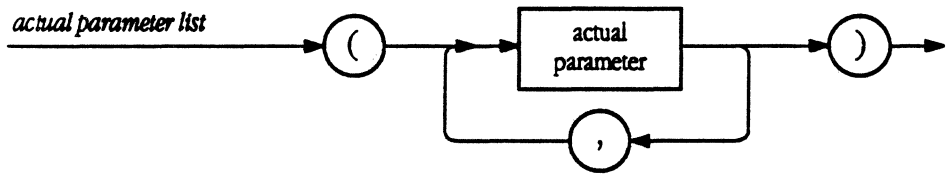
A function call executes the function denoted by the function identifier. A function identifier is any identifier that has been declared to denote a function, as described in Chapter 8.

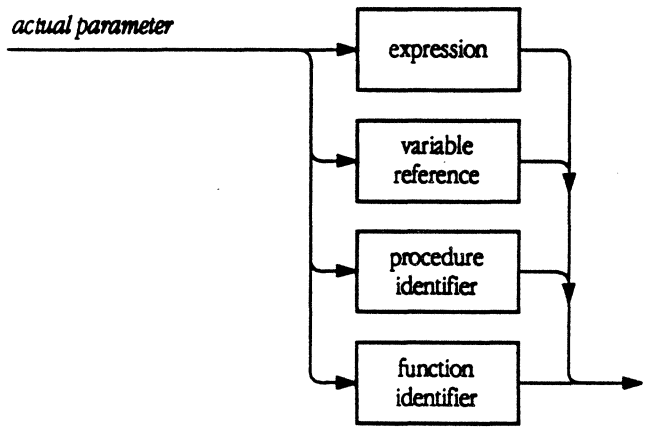
If the corresponding function declaration contains a list of formal parameters, then the function call must contain a corresponding list of actual parameters.

Each actual parameter is substituted for the corresponding formal parameter, according to these rules:

- The correspondence is established by the positions of the parameters in the lists of actual and formal parameters, respectively.
- The number of actual parameters must be equal to the number of formal parameters.
- The order of evaluation and binding of the actual parameters is unspecified.







Here are some examples of function calls:

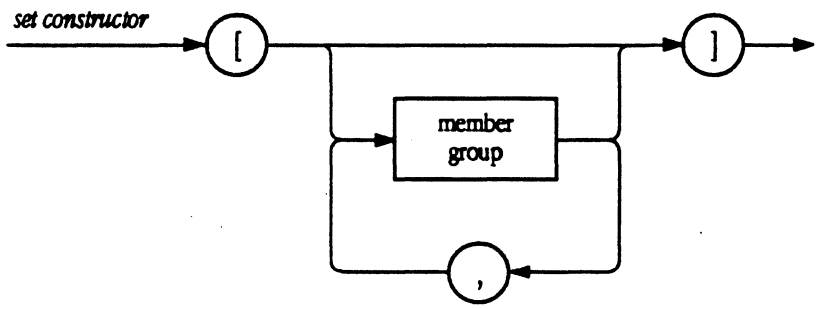
```
sum(a, 63)
gcd(147, k)
sin(x+y)
eof(f)
ord(f^)
```

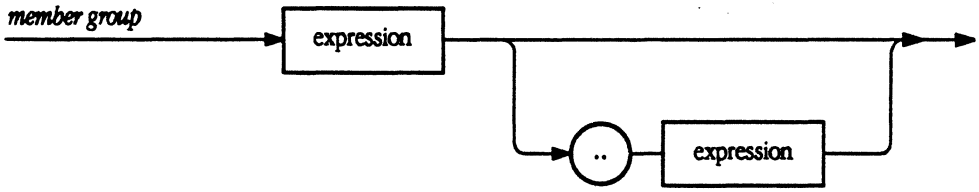
A special case of a function call is the method call. Method calls are discussed in detail in Chapter 12.

See Chapter 7 for a description of the procedure call statement.

Set constructors

A **set constructor** denotes a value of a set type and is formed by writing expressions within square brackets, `[ ]`. Each expression denotes a value of the set.





The notation `[]` denotes the empty set, which belongs to every set type. Any member group *x*`..`*y* denotes as set members the range of all values of the base type in the closed interval *x* to *y*.

If *x* is greater than *y*, then *x*`..`*y* denotes no members and `[`*x*`..`*y*`]` denotes the empty set.

All values designated in member groups in a particular set constructor must be of the same scalar type. This scalar type is the base type of the resulting set. If an `integer` value designated as a set member is outside the limits 0..2039, the results are unspecified.

Here are some examples of set constructors:

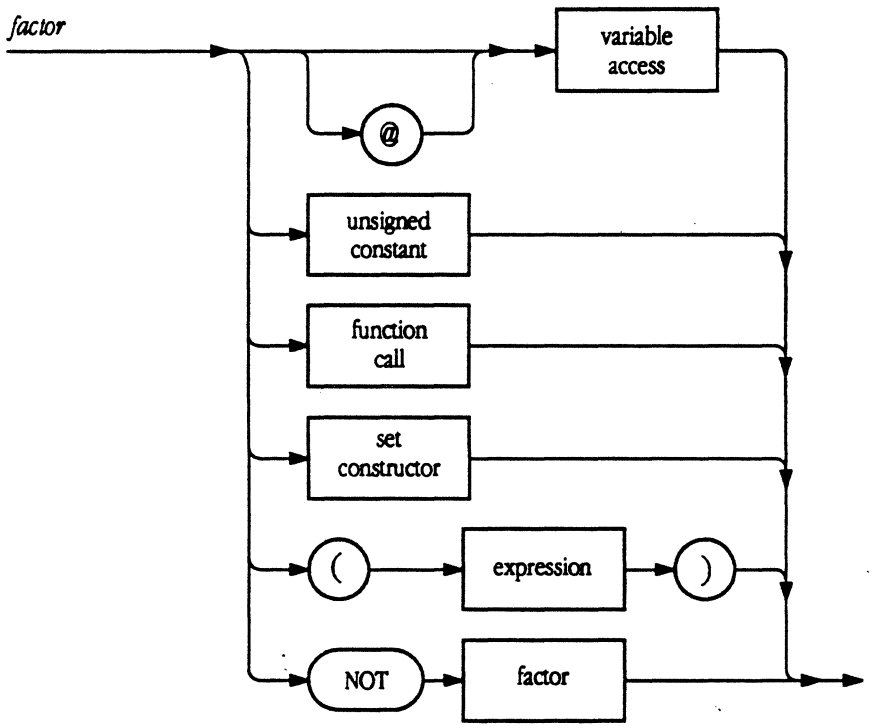
```
[red, c, green]
[1, 5, 10..k MOD 12, 23]
['A'..'Z', 'a'..'z', chr(xcode)]
```

Writing expressions

You build expressions from factors, terms, and simple expressions. These objects are described below.

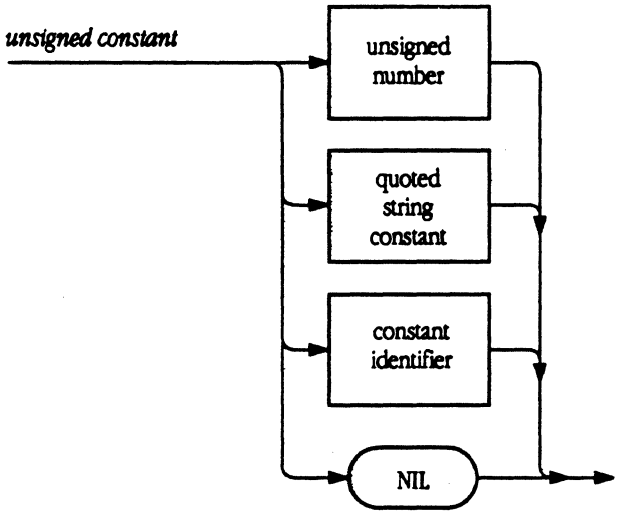
Factors

A **factor** may be any of the expressions shown in the following syntax diagram:



A function call activates a function and denotes the value returned by the function; functions are discussed in Chapter 8. A set constructor denotes a value of a set type, as described in Chapter 4.

An unsigned constant has the following syntax:

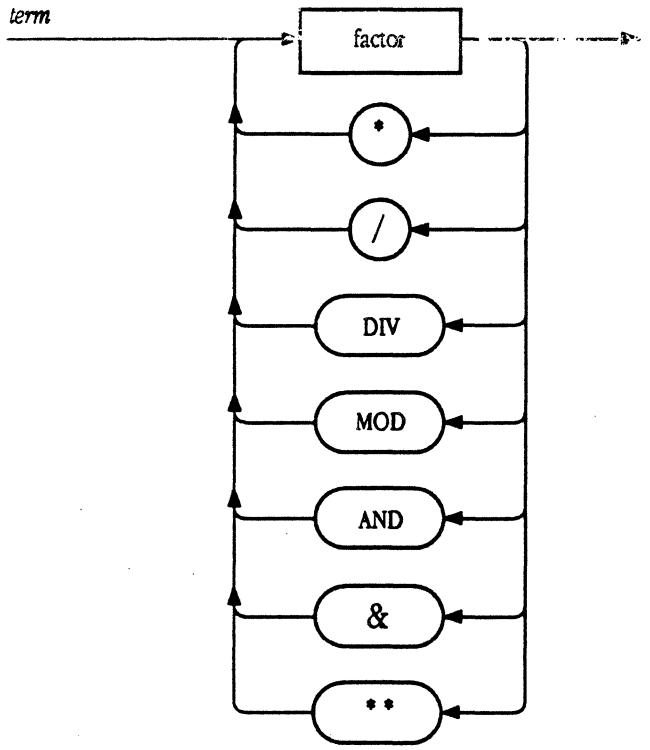


Here are some examples of factors:

```
x      {variable access}
@x     {pointer to a variable}
15     {unsigned constant}
(x+y+z)      {subexpression}
sin(x/2)     {function call}
['A'..'F', 'a'..'f']     {set constructor}
NOT p {negation of a boolean}
```

Terms

Terms apply the multiplying operators to factors:

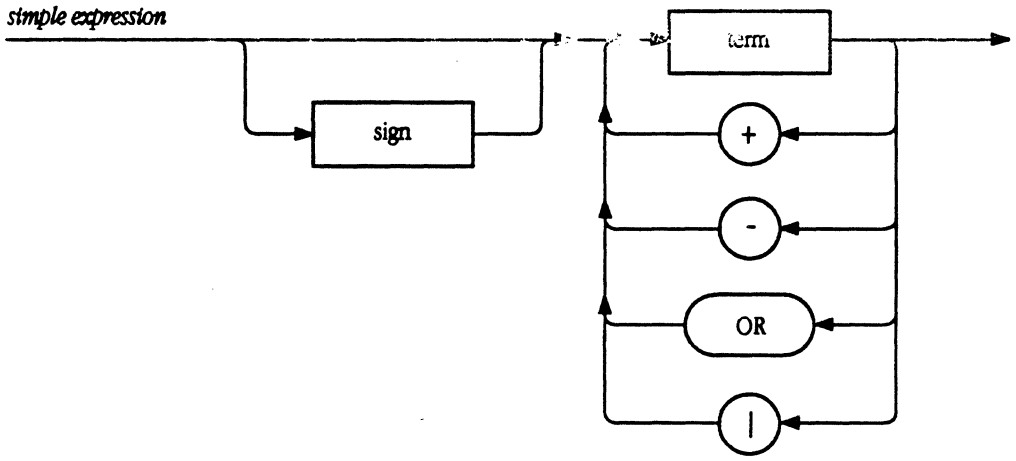


Here are some examples of terms:

```
x*y
i/(1-i)
p AND q
(x<=y) AND (y<z)
(i>0) & (a[i]=b)
```

Simple expressions

Simple expressions apply adding operators and signs to terms:

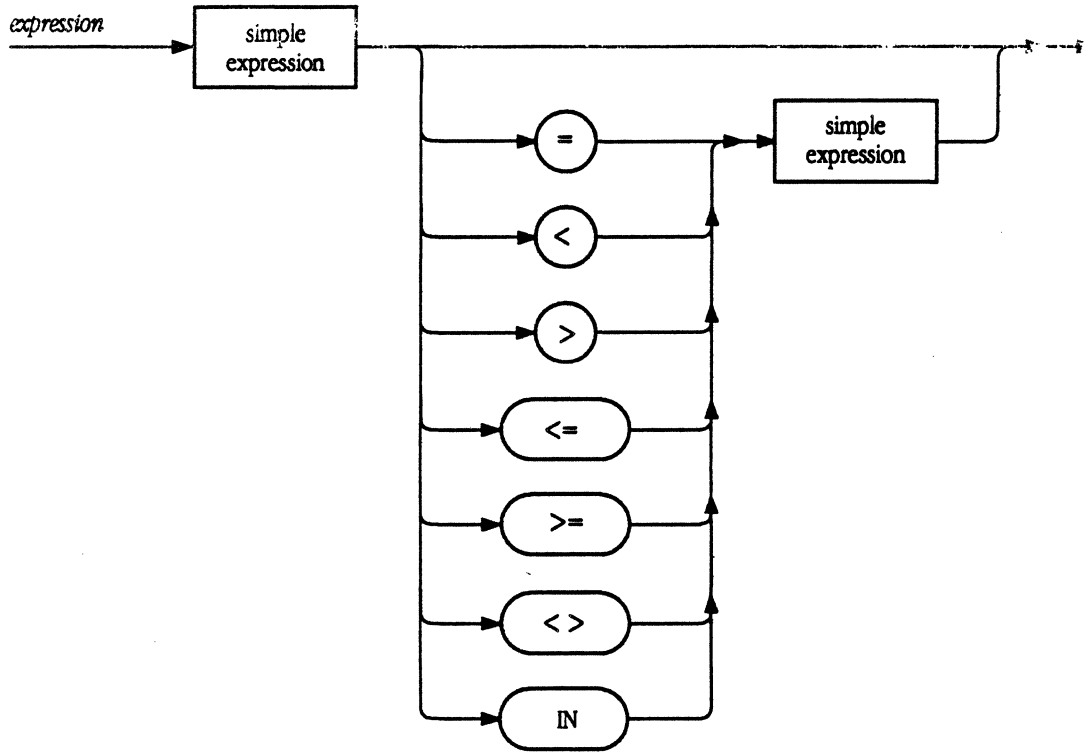


Here are some examples of simple expressions:

```
x+y
-x
hue1 + hue2
i*j + 1
```

Expression syntax

The syntax for an **expression** applies the **relational operators** to simple expressions:



Here are some examples of correctly written expressions:

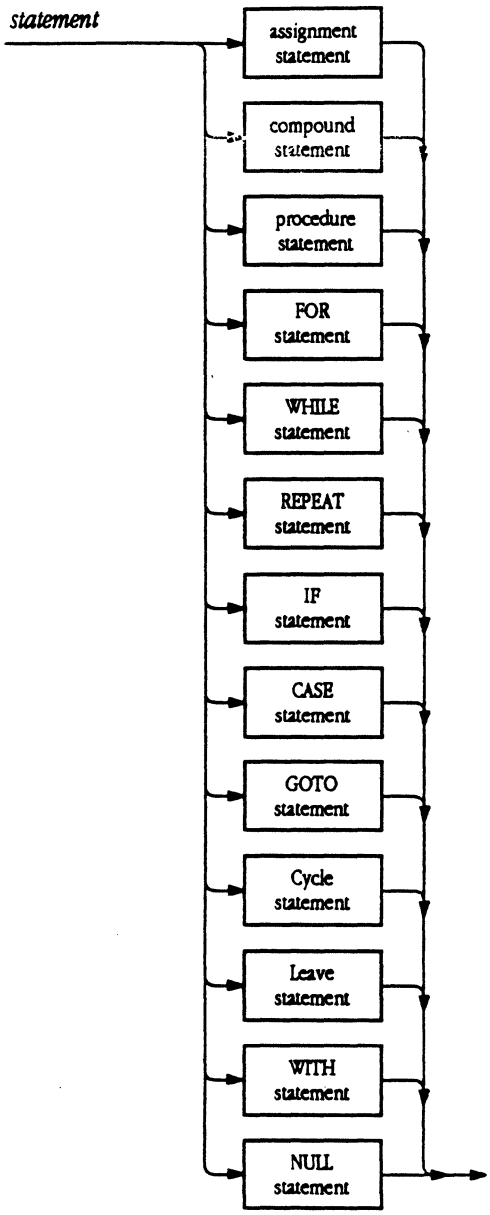
```
x=1.5
p<=q
p = q AND r
(i<j) = (j<k)
c IN hue1
```

Chapter 7 **Statements**

STATEMENTS ARE MADE UP OF EXPRESSIONS combined with certain reserved words. Statements describe algorithms and are executable. They perform the actual work of the Pascal program, doing such things as giving a value to a variable or providing conditional execution of other statements. ■

Contents

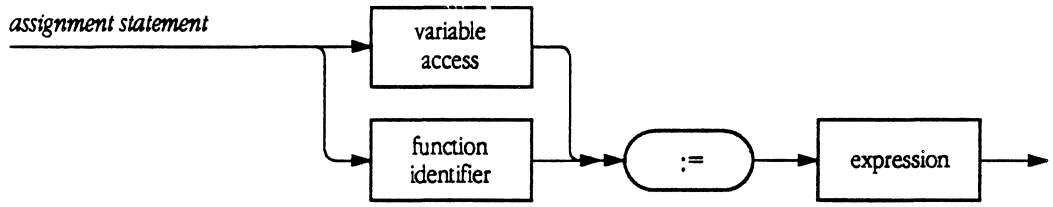
MPW Pascal has 13 statements:



These statements are discussed in this chapter.

Assignment statements

The assignment statement sets the value of a variable. The symbol := can be read as "set to." The statement is written:



The variable reference of the left side identifies a variable of any of the types except a file type. With most variables it is simply an identifying name, but in four cases it consists of a name followed by a qualification:

- If the variable is a string element, it is identified by the string name followed by the element's index number in brackets.
- If the variable is an array element, it is identified by the array name followed by one or more index values (one for each dimension of the array) enclosed in square brackets and separated by commas.
- If the variable is a record field, its name must be preceded by the name of its containing record and a period (unless the assignment statement is enclosed in a `WITH` statement).
- If the variable is a dynamic variable, it is identified by the name of its pointer followed by a caret.

In writing assignments, keep these rules in mind:

- A real-type variable may be set to the value of another real-type, an `integer` or `longint`, an `integer` subrange type, or an expression yielding `integer` or `real` results.
- A `longint` variable may be set to the value of an `integer`, an `integer` subrange type, or an expression yielding `integer` results. It may be set to the value of another `longint` or to an expression yielding `longint` results, provided the actual value does not exceed its declared size.
- A `boolean` variable may be set to the value of another `boolean` or to an expression yielding a `boolean` result.

- A `char` variable may be set to the value of another `char`, a `char` subrange type, or a string element.
- A scalar variable of subrange type may be set to the value of another scalar (or an expression yielding scalar results) of the host type, provided the actual value lies within its declared range.
- A user-defined scalar variable may be set to any of the values named in its declaration.
- A string variable may be set to the value of another string, provided its actual length does not exceed the variable's declared size.
- A set variable may take the value of another set variable or set constructor, provided they have the same base type.
- A whole array variable or record variable may be set to the value of another whole array or record variable of assignment-compatible type.
- A one-dimensional packed character array variable may be set to the value of a string constant (but not a string variable), provided its index range is the same as the string's length.
- Array elements (including elements that are arrays) and record fields (including fields that are records) act in assignments like ordinary variables of their declared types.

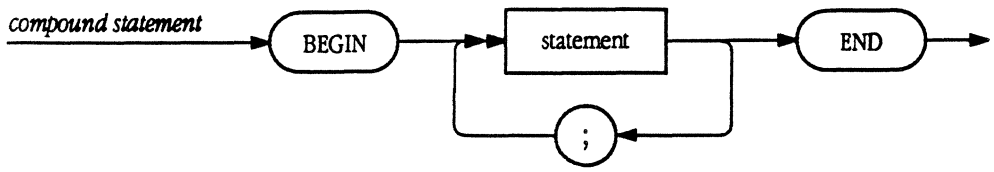
Here are some examples of assignment statements:

```
x := y+z
p := (1<=i) AND (i<100)
i := Sqr(k)-(i*j)
huel := [blue, succ(c)]
```

Compound statements

When writing Pascal programs, you often need to treat several statements as if they were one—for example, when they are all executed by a single control statement. To do this, you use the compound statement.

The body of every Pascal procedure, function, and main program consists of a single compound statement. To create a single compound statement out of a sequence of statements, preface the sequence with BEGIN and terminate it with END, separating the internal statements with semicolons.



The arrow coming back through the semicolon indicates that many statements may be placed between BEGIN and END, as long as they are separated by semicolons.

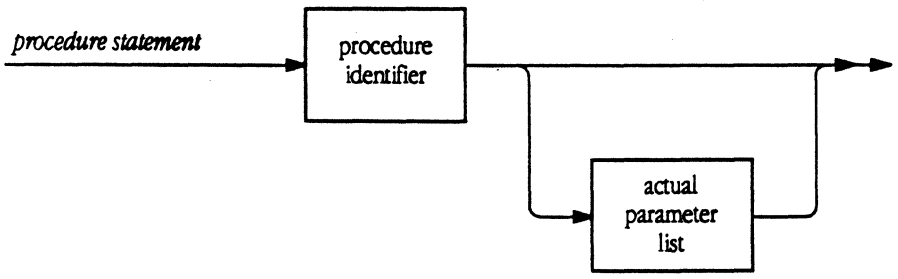
You can nest any number of BEGIN. . . END statements. Within any block, the Compiler will associate the last BEGIN with the first END, the next-to-last BEGIN with the second END, and so on. If you have written more BEGINs than ENDs, the Compiler will stop and display an error message.

Here is an example of a compound statement:

```
BEGIN
 z := x;
 x := y;
 y := z
END
```

Procedure statements

A procedure is called by writing its identifier in the source text, followed by its actual parameter list (if it has one) in parentheses. The parameters in a parameter list are separated by commas.



The identifier must be the same as the identifier used in the procedure or function declaration.

The parameter list in a procedure or function call contains the same number of formal parameters as were listed in the procedure or function declaration. Those in the declaration are called **formal parameters**; those in the calling statement, **actual** (or source) **parameters.** The values of the actual parameters are said to be passed to the formal parameters as part of the call.

The order and number of actual parameters in the call must match the order and number of formal parameters in the declaration. Each actual parameter must have the same type as the corresponding formal parameter, with these exceptions:

- Subrange types are equivalent to their base types.
- A formal parameter of type `longint` will accept an actual parameter of type `integer`.
- Formal parameters preceded by `univ` accept any actual parameter that occupies the same space in memory. For a full discussion of `univ`, see Chapter 9.

In addition, the actual parameters specified in any procedure or function call must follow these rules:

- Actual variable parameters must be variables. As opposed to value parameters, variable parameters cannot be constants, expressions, or elements of packed variables.
- The value of any actual string variable may be passed to any formal variable string parameter, regardless of length. However, if the declared maximum length of the formal parameter is longer then the declared maximum length of the actual parameter, you will get a Compiler error. You can avoid the problem by suspending range checking with the Compiler directive `$R-`.
- If the value of an actual parameter exceeds the range of a formal parameter (for instance, because the formal parameter is a subrange type), you will get an execution error unless you have suspended range checking with the Compiler directive `$R-`.

Here are some examples of procedure statements:

```
printheading
transpose(a, n, m)
bisect(fct, -1.0, +1.0, x)
```

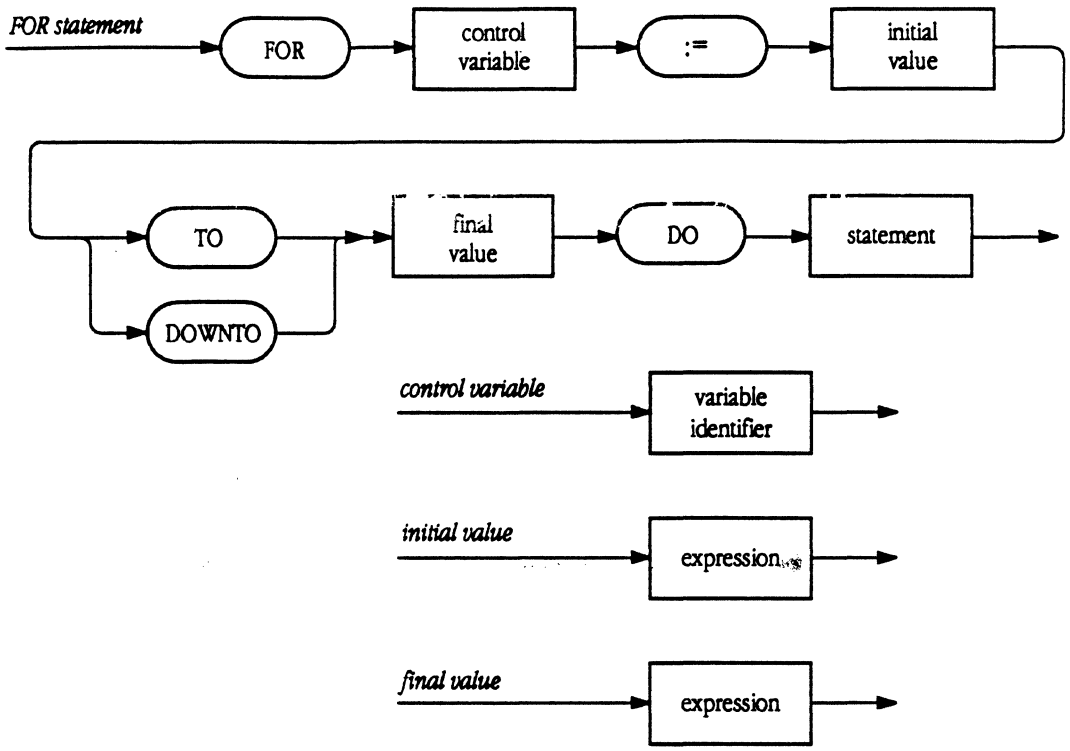
Repetition statements

Pascal provides three ways to execute the same program section repeatedly—the process called *looping*. But Pascal sets up the loop and exit routines for you; all you need to do is tell it the conditions for repetition. The repetition statements are the following:

- The `FOR` statement `FOR...DO` executes the same program section a given number of times. The number of executions may be constant or may be determined by the result of any scalar calculation.
- The `WHILE` statement `WHILE...DO` executes the same program section repeatedly as long as a given `boolean` expression is `true`. It evaluates the `boolean` control before each pass, including the first time; hence it can bypass the program section altogether.
- The `REPEAT` statement `REPEAT...UNTIL` also executes the same program section repeatedly as long as a given `boolean` expression is `true`. But it evaluates the `boolean` control after each pass; hence it executes the program section at least once.

FOR statements

The `FOR` statement requires a previously declared local variable of scalar type. It repeatedly increments or decrements the value of the variable, executing a section of your program each time. You define the starting and ending scalar values (which may be constant or calculated) and whether the `FOR` statement is to count upward or downward.



The control variable is the name of a scalar variable—`integer`, `char`, `boolean`, subrange, or user-defined. It cannot be an array or string element, a record field, or a dynamic variable. It must be declared in the block that contains the FOR statement. The FOR statement gives it a value before each pass through the program section it controls. Note that the value of this variable is accessible in the controlled section.

The initial and final value expressions must have the same scalar type as the variable. They may be simple constants or variables, or complex expressions containing operators and functions.

You write TO or DOWNTO, depending on whether the ordinality of the value of the second expression is higher or lower than the ordinality of the value of the first expression.

The statement controlled by the FOR statement can be a single other statement (such as an assignment or a procedure call) or a compound statement containing many other statements.

Observe these rules and cautions when writing any FOR statement:

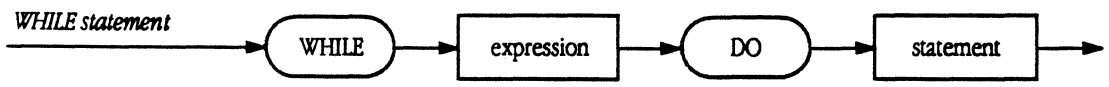
- The control variable must be a simple variable with local scope.
- If the control variable is a subrange type or user-defined scalar, it must be capable of accepting the initial and limit values as well as all values with an ordinality in between.
- Do not try to change the value of the control variable from within the FOR statement; doing so can have unpredictable results.
- Do not include the control variable in either of the limit expressions.
- After the FOR statement is finished, the value of the control variable may be unspecified.
- The limit expressions are evaluated just once, before the first pass. Changing them from within the FOR statement will not alter its behavior.
- If the limit expressions have equal value, the FOR statement will execute its controlled statement once.
- If the limit values are reversed—large limit less than small limit—the FOR statement will be skipped.

Here are some examples of FOR statements:

```
FOR i := 2 TO 63 DO IF a[i]>max THEN max := a[i]

FOR i := 1 TO n DO FOR j := 1 TO n DO
 BEGIN
  x := 0;
  FOR k := 1 TO n DO x := x+m1[i, k]*m2[k, j];
  m[i, j] := x
 END

FOR c := red TO blue DO q(c)
```

WHILE statements

The WHILE statement evaluates a `boolean` expression and then executes a statement if the expession is `true`. It repeats the execution, evaluating the expression before each pass, until the expression becomes `false`. The WHILE statement is written as follows:



The controlling expression must have `boolean` type; usually it is formed out of relational and logical operators.

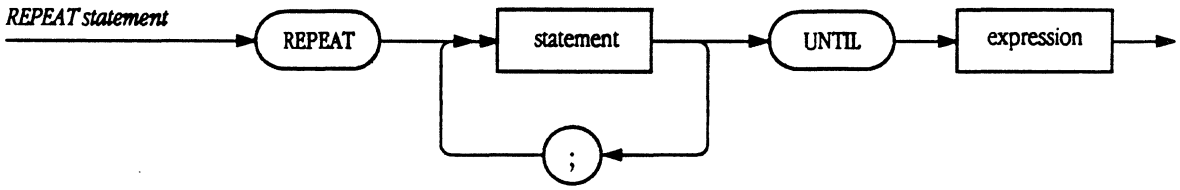
The statement controlled by `WHILE...DO` may be either a single statement or a compound `BEGIN...END` construction containing other statements.

Here are some examples of `WHILE` statements:

```
WHILE a[i]<>x DO i := i+1

WHILE i>0 DO
 BEGIN
  IF odd(i) THEN z := z*x;
  i := i DIV 2;
  x := sqr(x)
 END

WHILE NOT eof(f) DO
 BEGIN
  process(f^);
  get(f)
 END
```

REPEAT statements

The `REPEAT` statement behaves much like the `WHILE` statement, but it evaluates its `boolean` expression after executing the statements it controls. It is written as follows:



`REPEAT` and `UNTIL` create their own compound out of the statements they control; you do not need to use `BEGIN` and `END`.

The controlling expression must have `boolean` type; usually it is formed out of relational and logical operators.

- *Note:* With both `WHILE` and `REPEAT`, take care that the program statements they control include some practical means to change the expression, or to escape by means of a `GOTO` or `Leave` statement or `Exit` call. Otherwise your program can never terminate.

Here are some examples of REPEAT statements:

```
REPEAT
 k := i MOD j;
 i := j;
 j := k
UNTIL j = 0

REPEAT
 process(f^);
 get(f)
UNTIL eof(f)
```

Loop control: a comparison

The three repetition statements each have specific advantages and disadvantages in any given programming situation. Here are some of them.

The FOR statement automatically keeps track of which repetition it is executing, by changing the value of its control variable at the end of each pass. Thus you can use the control value to modify what your program does each time. For example, the control value can cause the repeated section to

- select a different element in an array each time by changing the index number
- call a different procedure each time by serving as the selector value for the CASE statement (described below)
- perform a different calculation each time by becoming a factor in an expression

The FOR statement is somewhat inflexible, however. You can change its number of repetitions only by terminating it with a GOTO or Leave statement.

The WHILE statement and REPEAT statement allow better control of the conditions under which they stop executing. The main difference between them is that the WHILE statement need not be executed at all, whereas the REPEAT statement executes at least once. Thus the WHILE statement is most useful when the condition controlling its execution may have already been satisfied; the REPEAT statement is most useful when the condition can be satisfied only by executing the statement.

The WHILE statement should also be used in cases where executing it under the wrong conditions could be detrimental, because it evaluates its control before each pass.

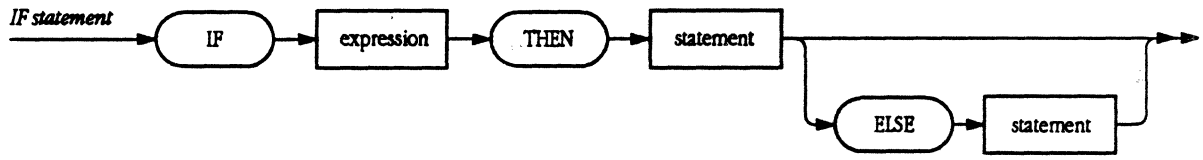
Conditional statements

Pascal provides two ways for your program to choose what to do next—the process sometimes called *branching:*

- The `IF` statement `IF...THEN...ELSE` evaluates a `boolean` expression and executes a controlled statement only if it is `true`. It can also be written to execute a second statement if the expression is `false`.
- The `CASE` statement `CASE...OF...OTHERWISE` executes one statement from a list, depending on the value of a scalar control expression.

IF statements

The `IF` statement executes a single controlled statement (which may be a compound `BEGIN...END` construction) if a `boolean` expression is `true`. You can add an optional `ELSE` part on the end that executes another (possibly compound) statement if it is `false`:



The controlling expression between `IF` and `THEN` must have `boolean` type; usually it is formed out of relational and logical operators.

Either or both controlled statements may be single statements or compound `BEGIN...END` constructions containing other statements. The only place you need to put a semicolon in an `IF` statement is within a compound `BEGIN...END` construction.

When executing an `IF` statement, Pascal performs these steps:

1. It evaluates the `boolean` expression.
2. If its value is `true`, Pascal executes the statement following `THEN` and exits the `IF` statement.
3. If its value is `false` and there is a statement after `ELSE`, Pascal executes it; otherwise, it exits the `IF` statements.

Here are some exampels of IF statements:

```
IF x<1.5 THEN z := x+y ELSE z := 1.5
IF p1<>nil THEN p1 := p1^.father
```

Nested IF statements

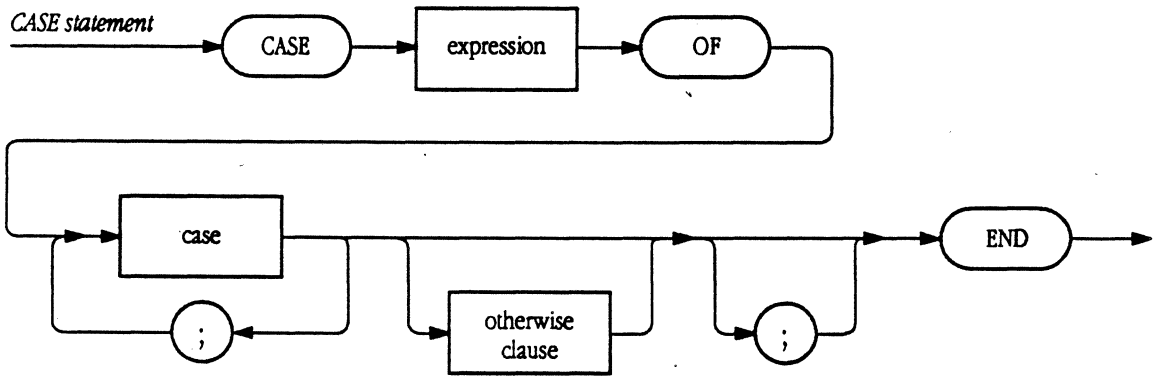
In any IF statement, the statement following the word ELSE can also be an IF statement and can contain its own ELSE clause. Thus an IF statement can be written to take different actions for each of several mutually exclusive conditions.

Pascal will evaluate boolean expressions only until a true one is found. You get maximum execution speed if you put the most probable conditions first.

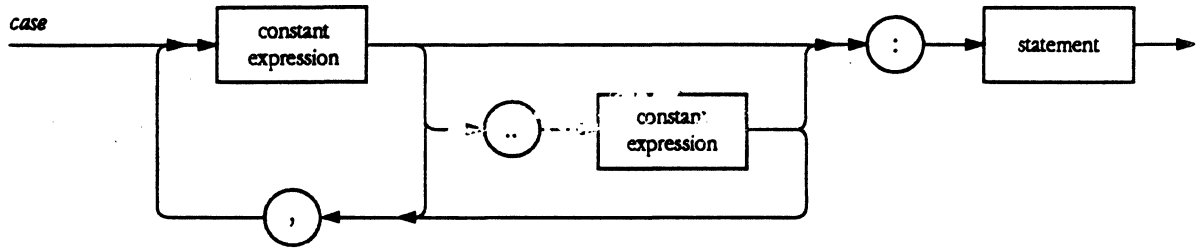
The statement following the word THEN can also be a nested IF statement, but this can create confusing source text. If it becomes unclear which ELSE matches which THEN, clarify the situation by using a compound BEGIN...END construction or appropriate indentation.

CASE statements

The CASE statement lets you write a list of alternative statements to be executed, associating a scalar constant with each one. When executing the CASE statement, Pascal evaluates a controlling scalar expression; if its value matches one of the constants, Pascal executes the corresponding statement. You can add an optional OTHERWISE part on the end that executes an additional statement if nothing was selected from the list. The CASE statement follows this syntax:



The clauses shown in the diagram have the following form:



The controlling expression may have any scalar type—`integer`, `char`, `boolean`, subrange, or user-defined. It should be capable of returning the value of any of the constants in the `CASE` clause.

The constant expressions in the `CASE` clause must have the same scalar type as the controlling expression.

Any of the controlled statements in the `CASE` clause or the default statement following `OTHERWISE` may be single statements or compound `BEGIN...END` constructions containing other statements.

Here are two examples of `CASE` statements:

```
CASE operator OF
 plus: x := x+y;
 minus: x := x-y;
 times: x := x*y
END

CASE i OF
 1:    x := sin(x);
 2:    x := cos(x);
 3, 4, 5: x := exp(x);
 OTHERWISE x := ln(x)
END
```

Control statements

The three repetition statements and two conditional statements described in this chapter, along with assignments and procedure calls, are flexible enough to handle almost all programming jobs. Occasionally, however, you may encounter a situation that demands immediate transfer or suspension of program execution. For these cases, Pascal provides five additional tools:

- the `GOTO` statement, which transfers control directly from one program statement to another
- the `Cycle` statement, which forces an immediate reiteration of a repetition statement loop
- the `Leave` statement, which immediately cancels a repetition statement loop
- the `Exit` procedure, which terminates any procedure, function, or whole program
- the `Halt` procedure, which stops program execution then and there

The `GOTO`, `Cycle`, and `Leave` statements are described below. The `Exit` and `Halt` procedures are described in Chapter 12.

GOTO statements

The `GOTO` statement transfers program execution to the beginning of any statement that is within the same procedure, function, or main program. Before you can use a `GOTO` statement, you must do two things:

- Declare a label for every `GOTO` destination in your program. Each label is a number of one to four digits. The label declaration consists of the reserved word `LABEL`, followed by one or more label numbers separated by commas. Label declarations are discussed under "Block Syntax" in Chapter 4.
- Write one of the declared destination labels, followed by a colon, in front of the statement that is the destination for each `GOTO` statement.

The `GOTO` statement is written

The unsigned integer is the destination label; it must not exceed four decimal digits.

Two more cautions apply to GOTO statements:

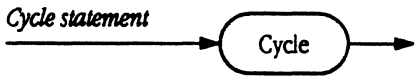
- The destination of any GOTO statement must be the beginning of a statement.
- Jumping to a statement that is within the structure of another statement (except with a compound statement that forms a program block) can have undefined effects, although the Compiler will not indicate an error.

Thus, every GOTO destination should be the beginning of a statement that is at the top level of nesting in a program block.

Here is an example illustrating the use of a GOTO statement:

```
BEGIN
 1234: Write ('Give me a number: ');
 Readln(n);
 IF n=0 THEN GOTO 1234
END
```

Cycle statements



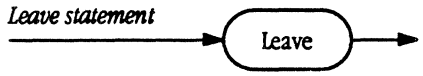
The `Cycle` statement passes program control to the end of the looping portion of the smallest `WHILE`, `REPEAT`, or `FOR` statement that encloses it. It is similar to the `continue` statement in the C language.

Here is an example illustrating the use of a `Cycle` statement. It calls the procedure `f` for all positive values of `a[i]`:

```
FOR i := 1 TO n DO
 BEGIN
  IF a[i]<=0 THEN Cycle;
  f(a[i])
 END
```

◆ *Note:* The word *Cycle* is not a reserved word. If you redefine it, you cannot use `Cycle` statements within the scope of your definition.

Leave statements



The `Leave` statement terminates the smallest `WHILE`, `REPEAT`, or `FOR` statement that encloses it, passing control to the next statement. It resembles the `break` statement in C.

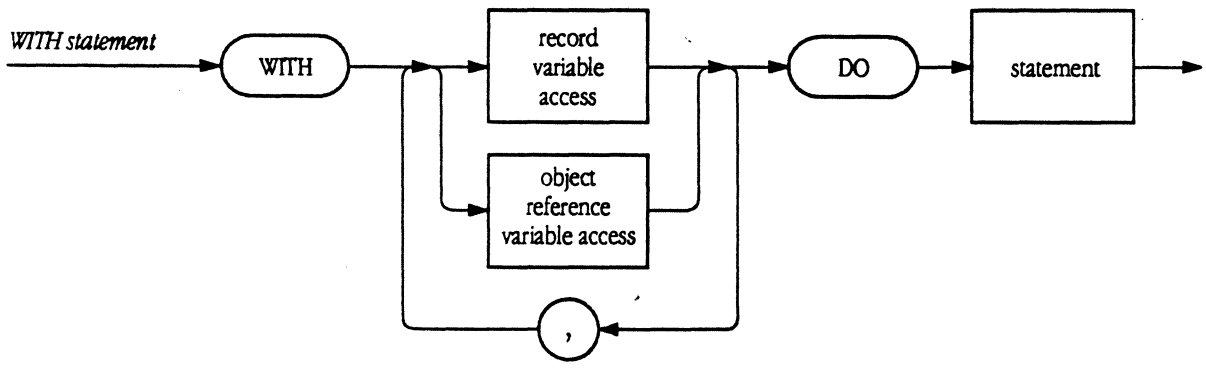
Here is an example illustrating the use of a `Leave` statement; in it, the `WHILE` statement terminates when the first `x` value of `a[i]` is found:

```
WHILE i<63 DO
 BEGIN
  IF a[i]=x THEN leave;
  i := i+1
 END
```

- *Note:* The word *Leave* is not a reserved word. If you redefine it, you cannot use `Leave` statements within the scope of your definition.

WITH statements

The `WITH...DO` statement provides a means by which the fields of specified records can be referenced using only their field identifiers. It has the following syntax:



Any number of record variable identifiers, including those of records that are fields of other records, may be listed between WITH and DO. The statement

```
WITH v1, v2, v3 DO s;
```

is equivalent to the group of WITH statements

```
WITH v1 DO
 WITH v2 DO
  WITH v3 DO s
```

The following rules govern the use of WITH...DO:

- When listing a record that is a field of another record, you must either list the containing record earlier or list that field in explicit form.
- WITH statements may be nested. The record variables "opened" by any WITH statement remain open in the nested statements.
- Where fields of different record variables have the same name, WITH accesses the field of that name in the record last listed, including redundant listings in nested statements. The identity of field names does not cause a Compiler error.
- Where a record field identifier is the same as a variable or other identifier declared outside the record, WITH accesses the field.
- Within a WITH statement, fields may still be identified explicitly, even though their record variables are listed. This feature can be used to resolve the ambiguity of identical field names.
- When used with variant record variables, WITH accesses the identifiers for their tag fields and all variant fields.

Here is an example of a WITH statement:

```
WITH date DO IF month=12 THEN
 BEGIN
  month := 1;
  year := year+1
 END
 ELSE month := month+1
```

NULL statements

NULL statements are statements that don't contain anything. This simply means that whenever Pascal syntax calls for a statement, you can omit it. It also means that when a program contains an unnecessary semicolon, the Pascal Compiler considers the semicolon to be separating a null statement from another statement. The result is two statements where you only intend one. Most of the time, this is harmless, but it occasionally causes an error when only one statement is allowed.

Chapter 8 **Procedures and Functions**

WRITING PROCEDURES AND FUNCTIONS IN YOUR PROGRAM lets you nest additional blocks inside the main program block.

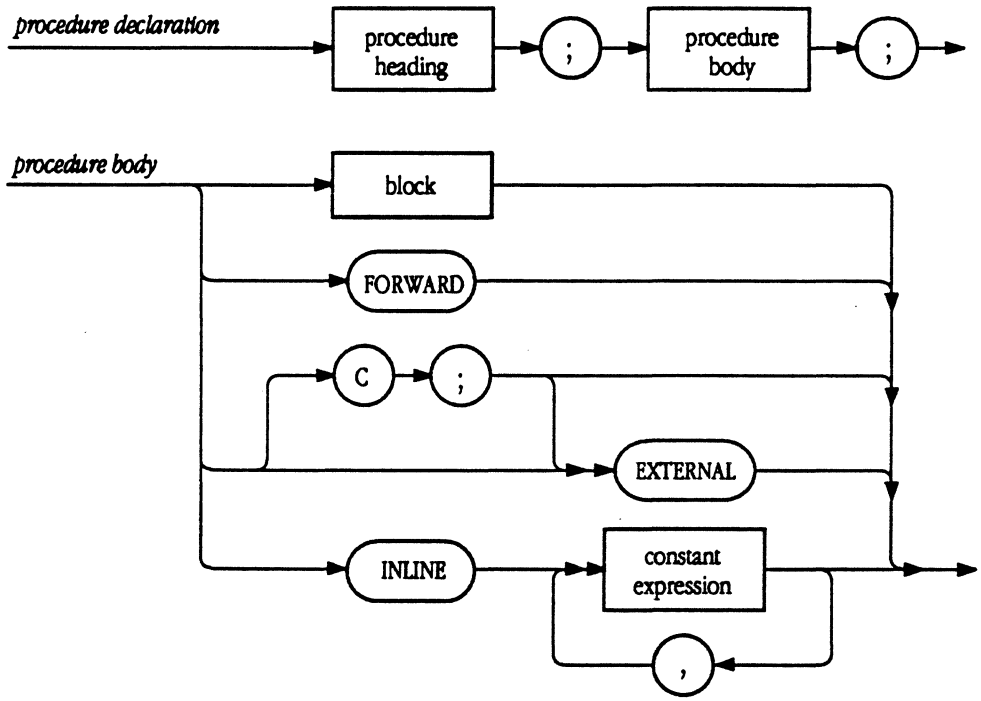
Each procedure or function declaration consists of a heading followed by a block. These are the principal differences between procedures and functions:

- Their heading formats are different.
- A procedure block is activated by a procedure call statement, as described in Chapter 7; a function block is activated by the evaluation of an expression that contains its call.
- A function returns a value to the expression that calls it. ■

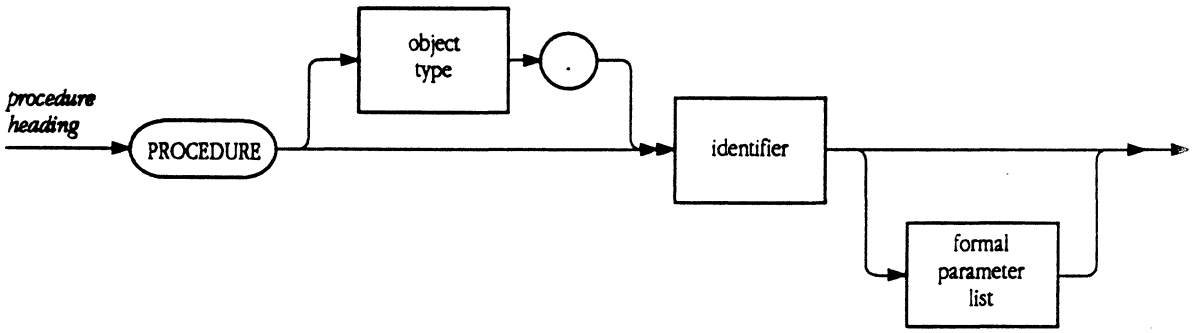
Contents

Procedure declarations

A procedure declaration associates an identifier with a block, so that part of the program can be activated by a procedure statement.



The procedure heading specifies the identifier for the procedure and its formal parameters (if any).



An object type is only given if this is a method declaration. A method declaration is a special type of procedure declaration made as part of an object type declaration. Detailed information on method declarations is given in Chapter 12.

The syntax for a formal parameter list is shown later in this chapter.

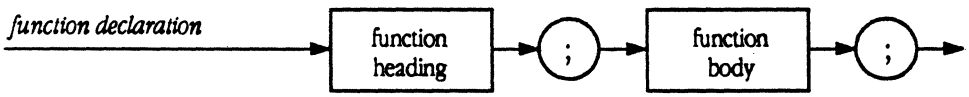
A procedure is activated by a procedure statement, as defined in Chapter 7, which gives the procedure's identifier and any actual parameters required by the procedure. The statements to be executed upon activation of the procedure are specified by the statement part of the procedure's block. If the procedure's identifier is used in a procedure statement within the procedure's block, the procedure is executed recursively.

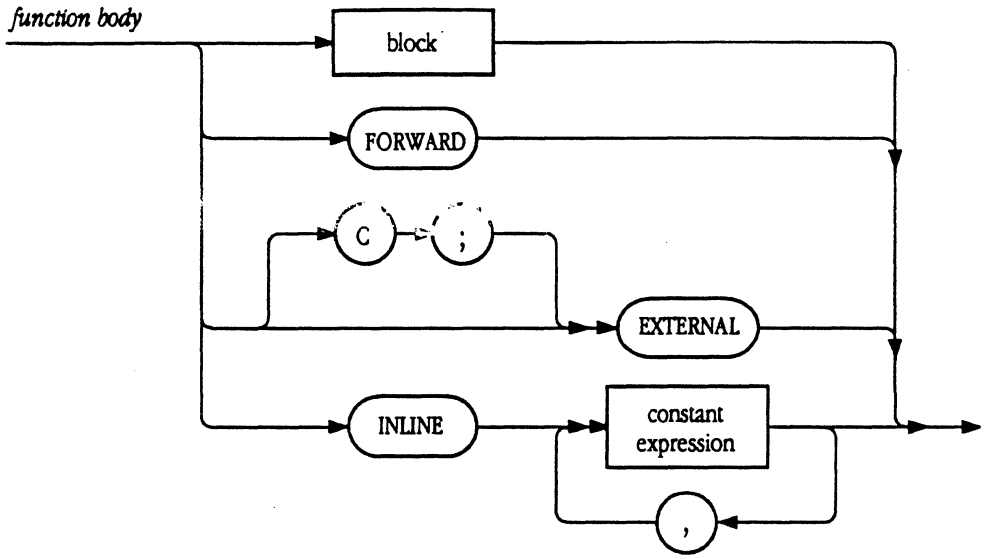
Here is an example of a procedure declaration:

```
PROCEDURE Summation (n: integer; a: intarray; VAR sum: longint);
 VAR i: integer;
 BEGIN {Summation}
 Sum := 0;
  FOR i := 1 TO n DO sum := Sum+a[i]
 END; {Summation}
```

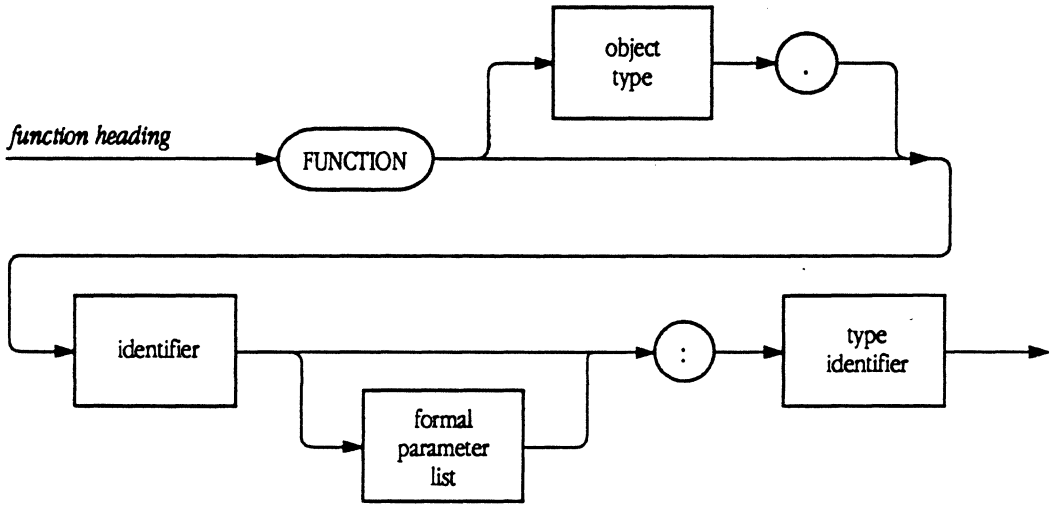
Function declarations

A function declaration serves to define a part of the program that computes and returns a value. The return value can be of any type.





The function heading specifies the identifier for the function, the formal parameters (if any), and the type of the function result.



The syntax for a formal parameter list is given later in this chapter.

A function is activated by the evaluation of a function call (see Chapter 6), which gives the function's identifier and any actual parameters required by the function. The function call usually appears as an operand in an expression. The expression is evaluated by executing the function and replacing the function call with the value returned by the function.

Function calls can also be used with variable qualifiers to identify a variable to which a value is assigned. In that case, the function is executed, the qualifiers are applied to the result of the function, and the result of the expression is assigned to the variable thus located. It is not specified whether the expression or the function is evaluated first.

The statements to be executed upon activation of the function are specified by the statement part of the function's block. This block should contain at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value assigned. If no such assignment statement exists or if it exists but is not executed, the value returned by the function is unspecified. If the return value of the function is a structured type, you can assign values to components alone or to the entire structure.

If the size of the return value of the function is no more than four bytes, the return value itself is placed on the stack. If the return value is longer than four bytes, a pointer is placed on the stack and code is generated so that the return value is obtained through the pointer.

If the function's identifier is used in a function call within the function's block, the function is executed recursively.

- *Note:* If the return value of a function is a record type, a pointer to a record type, or an object type, you cannot use a WITH statement to assign values to the fields of the record or object. The Compiler will interpret use of the function's identifier in the WITH statement as a function call.

Here are some examples of function declarations:

```
FUNCTION max(a: vector; n: integer): extended;
 VAR x: extended; i: integer;
 BEGIN
  x := a[1];
  FOR i := 2 TO n DO if x < a[i] then x := a[i];
  max := x
 END;
FUNCTION power(x: extended; y: integer): extended; { y >= 0}
 VAR w, z: extended; i: integer;
 BEGIN
  w := x; z := 1; i := y;
  WHILE i > 0 DO BEGIN
   {z*(w**i) = x**y}
   IF odd(i) THEN z := z*w;
   i := i DIV 2;
   w := sqr(w)
  END;
  {z = x**y}
  power := z
 END;
FUNCTION RelRect(BaseRect: Rect; top, left, bot, right: integer): Rect;
 BEGIN
  RelRect.top := BaseRect.top + top;
  RelRect.left := BaseRect.left + left;
  RelRect.bot := BaseRect.bot + bot;
  RelRect.right := BaseRect.right + right
 END;
---
newRect := RelRect(oldRect, 10, 10, 20, 20);
```

Procedure and function directives

In place of the block in a procedure or function declaration, you can write the following directives:

- FORWARD lets you use the procedure or function immediately but postpone defining the block to a later part of your program.
- EXTERNAL and C let you link a C or assembly-language routine to your program, which will be executed as the procedure or function's block.
- INLINE lets you write actual assembly-language instructions to be executed in place of the block.

These directives are described below.

The FORWARD directive

A procedure or function declaration containing the directive FORWARD instead of a block is called a **forward declaration.** Somewhere after the forward declaration but in the same block, the procedure or function is defined by a defining declaration, a declaration that uses the same identifier and includes a block. The formal parameter list may be repeated in the defining declaration; but if you repeat the formal parameter list, it must be identical to the list in the forward declaration. The forward declaration and the defining declaration must be local to the same block but need not be contiguous; that is, other procedures or functions can be declared between them and can call the procedure that has been declared forward. This permits **mutual recursion.**

The forward declaration and the defining declaration constitute a complete declaration of the procedure or function. The procedure or function is considered to be declared at the place of the forward declaration.

Here is an example of a forward declaration that permits mutual recursion:

```
PROCEDURE walter(m, n: integer);     {forward declaration}
 FORWARD;
PROCEDURE clara(x, y: real);
 BEGIN
  ---
  walter(4, 5);   {OK because walter is forward declared.}
  ---
 END;
PROCEDURE walter;        {defining declaration}
 BEGIN
  ---
  clara(8.3, 2.4);
  ---
 END;
```

Forward procedures and functions may not be written in the interface part of a unit.

The EXTERNAL and C directives

A procedure or function declaration containing the directive EXTERNAL instead of a block defines the Pascal interface to a separately assembled or compiled routine, such as a procedure code module in MPW assembly language. The external code must be linked with the compiled Pascal host program before execution; see the Linker instructions in the *Macintosh Programmer's Workshop 3.0 Reference* for details. Pascal and C calling conventions are described in Appendix F.

When you use the C directive in addition to EXTERNAL, the parameters (and function return values) are automatically arranged according to C language standards. As with other external routines, C-declared procedures and functions have no body; they are linked with C Compiler output by the Linker.

The C directive causes the Compiler to

- push parameters onto the stack in reverse order
- push all scalars as `longint` values and all real values as `extended` values
- expect function return values in register D0 (D0, D1, and A0 for `extended` results)

◆ *Note:* For nonreal results longer than four bytes, the address of the result is returned in register D0. In that case, the Compiler generates code to copy the result into the caller's space before continuing.

Here are two examples of external procedure declarations:

```
PROCEDURE MakeScreen(index: integer); EXTERNAL;
PROCEDURE Allen(howl: string); C; EXTERNAL;
```

In these examples, `MakeScreen` is an external procedure that must be linked to the host program before execution. `Allen` is a C procedure that must be linked to the host program before execution.

It is the programmer's responsibility to ensure that the external procedure or function is compatible with the EXTERNAL and C declarations in the Pascal program; the Linker does not check for compatibility.

External procedures and functions may not be written in the interface part of a unit.

The INLINE directive

The INLINE directive allows you to write explicit hexadecimal MC680x0 machine instructions in place of the block. The code is expressed in constants or constant expressions.

When a normal procedure or function is called, the Compiler generates code that pushes the procedure's arguments on the stack (along with two or four bytes for a return value, if this is a function) and then generates an assembly-language JSR (Jump to SubRoutine) to call the procedure, as explained in Appendix F. When you use a procedure declared INLINE, the Compiler generates code (in place of the JSR) from the constants following the word INLINE.

Each constant (or constant expression) represents exactly one machine-instruction word in the code generated by the Compiler. The code is generated in the order of the constants. Take care that you observe the proper rules for adjusting the stack, saving registers, and so on. These are documented in *Inside Macintosh* and Appendix F.

This facility is intended for writing small routines and Macintosh ROM routine calls. If you want to use large machine code routines, though, create an external procedure instead.

Unlike the FORWARD and EXTERNAL directives, no block is ever defined in an INLINE directive. INLINE can also be used in the interface part of a unit. In that case, there is still no block for the procedure in the corresponding implementation part.

Here is an example of a procedure declared INLINE:

```
PROCEDURE trap (Tos: longint); INLINE $A9ED;
```

The @ operator cannot be used to generate a pointer to an INLINE routine.

Parameters

Procedure and function declarations may have any or all of four kinds of formal parameters:

- value parameters
- variable parameters
- procedural parameters
- functional parameters

When writing a formal parameter list, you distinguish the four kinds as follows:

- A parameter group preceded by VAR is a list of variable parameters.
- A parameter group without a preceding VAR is a list of value parameters.
- A procedure heading denotes a procedural parameter.
- A function heading denotes a functional parameter.

A formal parameter list may be part of a procedure declaration or function declaration, or it may be part of the declaration of a procedural or functional parameter.

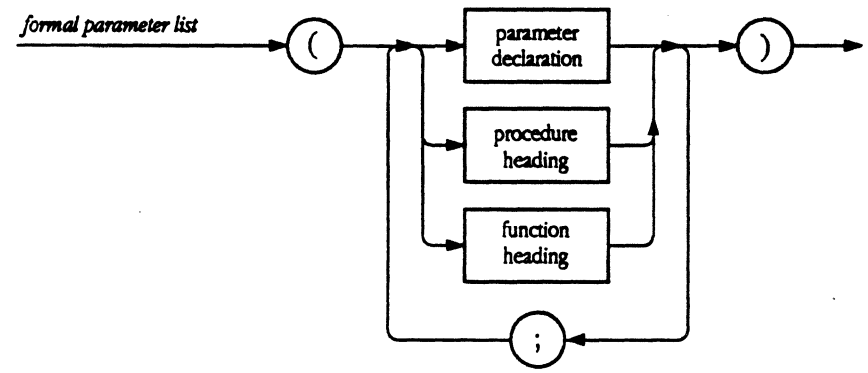
- *Note:* The types of formal parameters are denoted by type identifiers, so you cannot define a new type in a parameter list. In other words, only a simple identifier can be used to denote a type in a formal parameter list. To use a type such as `PACKED ARRAY [0..255] OF char` as the type of a parameter, you must first declare a type identifier for this type:

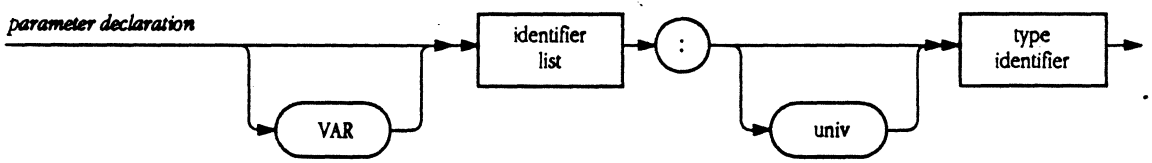
  ```
  TYPE charray = PACKED ARRAY[0..255] OF char;
  ```

  The identifier `charray` can then be used in a formal parameter list to denote the type.

If a formal parameter list is part of a procedure declaration or function declaration, it declares the formal parameters of the procedure or function. Each parameter so declared is local to the procedure or function being declared and can be referred to by its identifier in the block associated with the procedure or function.

If the list is part of the declaration of a procedural or functional parameter, it declares the formal parameters of the procedural or functional parameter. In this case, there is no associated block and the identifiers of parameters in the formal parameter list are significant only to the extent that they indicate the format and number of parameters.





- *Note:* The word `FILE` (for an untyped file) is not allowed as a type identifier in a parameter declaration, because it is a reserved word. To use a parameter of this type, declare some other identifier for the type `FILE`. For example,

```
TYPE phyle = FILE;
```

  The identifier `phyle` can then be used in a formal parameter list to denote the type `FILE`.

Value parameters

A formal value parameter acts like a variable local to the procedure or function, except that it gets an initial value from the actual parameter in the corresponding position in the actual parameter list.

- *Note:* At run time, the procedure makes a copy of each actual parameter value that is longer than four bytes in its own local variable space.

No changes made to a formal value parameter change the value of whatever is in the corresponding position in the actual parameter list.

For a value parameter, the corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of a file type or of any structured type that contains a file type.

The actual parameter must be assignment-compatible with the type of the formal value parameter. However, you can override this restriction by declaring the parameter as `univ`, as described later in this chapter.

Variable parameters

Variable parameters are used when a value must be passed back from a procedure or function to the calling program.

The corresponding actual parameter in a procedure statement or function call must be a variable access, as defined in Chapter 5. The formal variable parameter denotes the actual variable during the entire activation of the procedure or function, so any changes to the value of the formal variable parameter are reflected in the actual parameter.

Within the procedure or function, any access of the formal variable parameter is an access of the actual parameter itself. The type of the actual parameter must be identical to that of the formal variable parameter. However, you can override this restriction by declaring the parameter as `univ`, as described later in this chapter.

File types must be passed as variable parameters.

- *Note:* If the access of an actual variable parameter involves indexing an array, finding the identified variable of a pointer, or finding the field of a record or an object, these actions are executed before the activation of the procedure or function. If the variable is in a relocatable block of the heap, compaction of the heap can cause the original object to be moved, which yields unpredictable results.

Byte-aligned fields of packed structures are valid as variable parameters.

Procedural parameters

When the formal parameter is a procedure heading, the corresponding actual parameter in a procedure statement or function call must be a procedure identifier. The identifier in the formal procedure heading represents the actual procedure during execution of the procedure or function receiving the procedural parameter.

Here are some examples of procedural parameters:

```
PROGRAM passProc;
 VAR i: integer;
 PROCEDURE a(PROCEDURE x);     {x is a formal procedural}
  BEGIN       {parameter.}
   write('About to call x ');
   x  {Call the PROCEDURE passed as}
  END;        {parameter.}
 PROCEDURE b;
  BEGIN
   write('In PROCEDURE b')
  END;
 FUNCTION c(PROCEDURE x): integer;
  BEGIN
   x; {Call the PROCEDURE x, passed as}
   c:=2       {formal procedural parameter.}
  END;
 BEGIN
  a(b);      {Call a, passing b as parameter.}
  i:= c(b)   {Call c, passing b as parameter.}
 END.
```

If the actual procedure and the formal procedure have formal parameter lists, the formal parameter lists must be compatible, as described in Chapter 7. However, only the identifier of the actual procedure is written as an actual parameter; no parameters are given for the actual procedure.

Here is an example of procedural parameters with their own formal parameter lists:

```
PROGRAM test;
 PROCEDURE xAsPar(y: integer);
  BEGIN
   writeln('y=', y)
  END;
 PROCEDURE callProc(PROCEDURE xAgain(z: integer));
  BEGIN
   xAgain(1)
  END;
 BEGIN   {body of program}
  callProc(xAsPar) {Note only the PROCEDURE identifier is given.}
 END.
```

If the procedural parameter, upon activation, accesses any nonlocal entity (by variable access, procedure statement, function call, or label), the entity accessed must be one that was accessible to the procedure when the procedure was passed as an actual parameter. To see what this means, consider a procedure `Proc` that is local to another procedure, `firstPasser`.

Suppose that the following sequence takes place:

1. `FirstPasser` is executing.
2. `FirstPasser` calls a procedure named `firstReceiver`, passing `Proc` as an actual parameter.
3. `FirstReceiver` calls `secondReceiver`, again passing `Proc` as an actual parameter.
4. `SecondReceiver` calls `Proc` (first execution of `Proc`).
5. `SecondReceiver` calls `thirdReceiver`, again passing `Proc` as an actual parameter.
6. `ThirdReceiver` calls `firstPasser` (indirect recursion) and passes `Proc` to `firstPasser` as an actual parameter.
7. `FirstPasser` (executing recursively) calls `Proc` (second execution of `Proc`).

Thus the procedure `Proc` is called first from `secondReceiver` and then from the second (recursive) execution of `firstPasser`.

Suppose that `Proc` uses a variable access `pVar` and `pVar` is not local to `Proc`, and suppose that each of the other procedures has a local variable named `pVar`.

Each time `Proc` is called, which `pVar` does it access? The answer is that in each case, `Proc` accesses the `pVar` that is local to the first execution of `firstPasser`—that is, the `pVar` that was accessible when `Proc` was originally passed as an actual parameter.

Procedure pointers

The `@` operator can create procedure pointers. See "The @ Operator in a Procedure or a Function" in Chapter 6 for details on procedure pointers.

Functional parameters

When the formal parameter is a function heading, the actual parameter must be a function identifier. The identifier in the formal function heading represents the actual function during the execution of the procedure or function receiving the functional parameter.

Functional parameters are exactly like procedural parameters, with the additional rule that corresponding formal and actual functions must have identical result types.

Univ parameters

When the word `univ` is given before the type identifier in the formal parameter list, the corresponding item in an actual parameter list can be of any type that is the same size as the formal parameter's type.

Here is an example of a `univ` parameter:

```
TYPE
 ptr1 = ^char;
 ptr2 = ^integer;

VAR
 four: longint;
 pInt: ptr2;

PROCEDURE RealAddr(virt: longint; rAddr: univ ptr1);
---
RealAddr(v, pInt);   {pInt can be a pointer to a type}
             {other than char, or can be}
RealAddr(v, four);    {any other four-byte type.}
```

Parameter list compatibility

Parameter list compatibility is required of the parameter lists of corresponding formal and actual procedural or functional parameters.

Two formal parameter lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters match if one of the following is true:

- They are both value parameters of identical type.
- They are both variable parameters of identical type.
- The formal parameter has `univ` before its type, and the actual parameter is a value or variable of the same size. The parameters must still be both value parameters or both variable parameters.
- They are both procedural parameters with compatible parameter lists.
- They are both functional parameters with compatible parameter lists and identical result types.

Chapter 9 **Programs and Units**

THE PASCAL BLOCKS DISCUSSED IN CHAPTER 3 are assembled into **programs** and **units.** The principal difference between the two is that a program is complete and executable; a unit resembles a program but cannot be executed by itself. Both programs and units are separately compiled. Their object files are then combined by the Linker to form a single executable object file. This process is described in the *Macintosh Programmer's Workshop 3.0 Reference.*

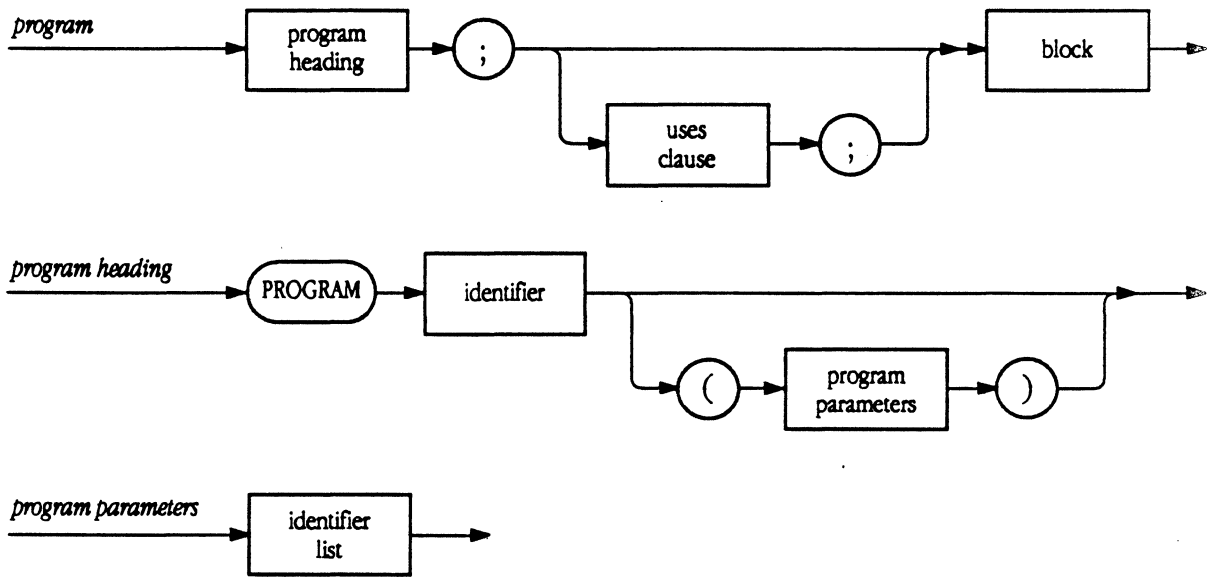
There are several reasons for using units in Pascal programming:

- They help modularize large programs.
- They make common declarations and blocks easily available to more than one program.
- They can be used to maintain the privacy of sections of a source text. ■

Contents

Program syntax

A Pascal program consists of a heading, an optional USES clause, and a block. (The USES clause is discussed later in this chapter.)



The occurrence of an identifier immediately after the reserved word PROGRAM declares it as the program's identifier.

- *Note:* Program-parameters, as described by Jensen and Wirth and the ANSI Standard, are ignored by MPW Pascal.

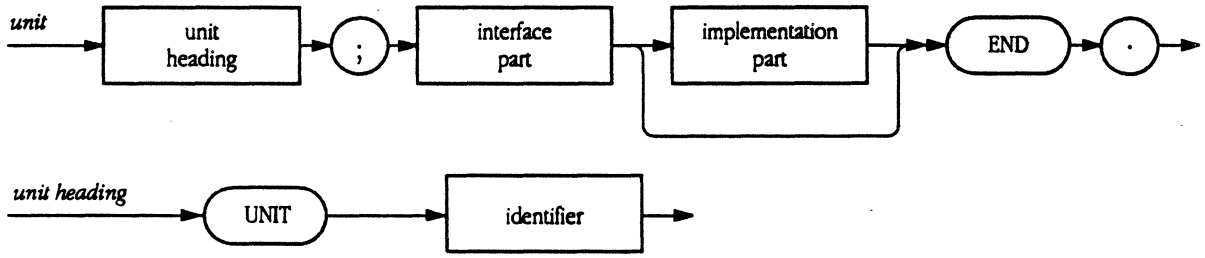
Segmentation

The code of every program's main body is always placed in a runtime segment whose name is `Main` (capitalization of the name *Main* is significant). Any other program block can be placed in a different segment by using the `$S` Compiler command described in Chapter 15. If no `$S` command is used in the program, all program code is placed in the `Main` segment.

By default, code copied from units is also placed in the `Main` segment. The code of any entire unit, or of any procedure or function within a unit, can be placed in one or more different segments by using the `$S` Compiler command in the unit's source text. (Procedures and functions are described in Chapter 8.)

Unit syntax

The syntax for writing a unit is

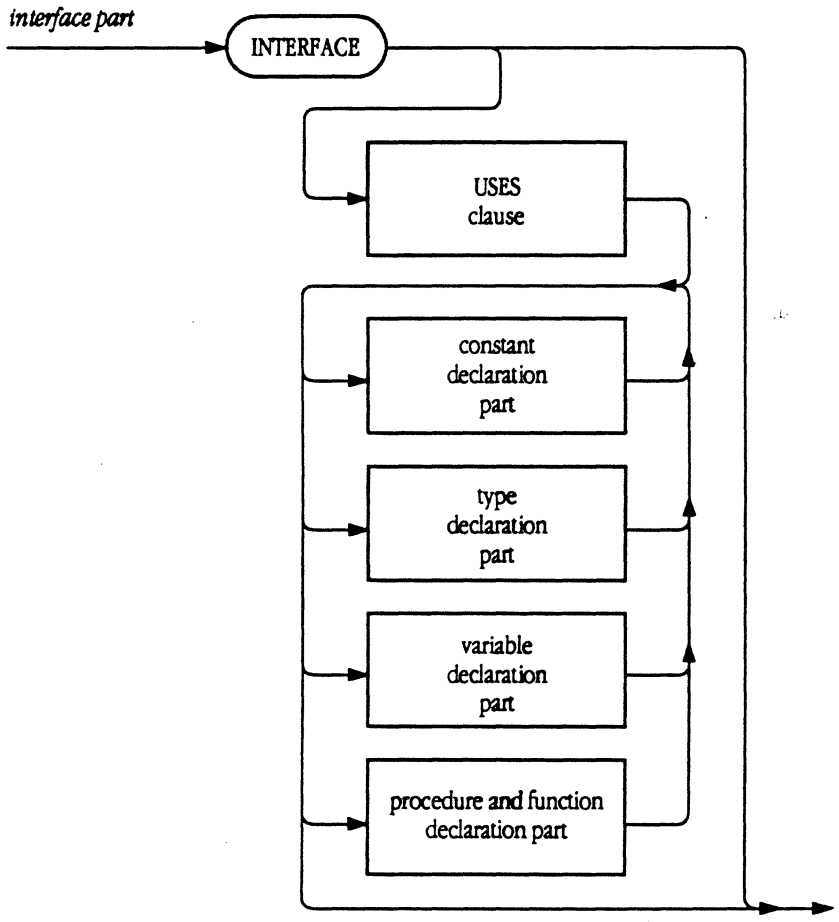


The **interface part** of a unit declares constants, types, variables, procedures, and functions that are "public"—that is, available to the **host program** (which may be another unit). In other words, the scope of the public entities is the entire host program. It can access these entities just as if they had been declared in its source text.

You declare procedures and functions in the interface part by giving only the procedure or function name, parameter specifications, and function result type. In other words, you give only the part that defines how the procedure or function is called. You declare methods in object type declarations the same way, except that you also specify `OVERRIDE` where appropriate.

If INLINE or C directives are used within routines in the unit, the directives must also appear in the interface to the unit. Otherwise, each piece of header information in the interface is treated like a FORWARD declaration when the unit is compiled.

Variables and routines that appear in the interface are global. The entire unit is within the scope of the block in which the USES clause that references the unit appears.



The optional **implementation part,** which follows the last declaration in the interface part, declares any constants, types, variables, procedures, or functions that are "private"—that is, not available to the host program. Private procedures and functions are declared like procedures and functions in programs, with a procedure or function heading and a body. For further information about declaring procedures and functions, see Chapter 8.

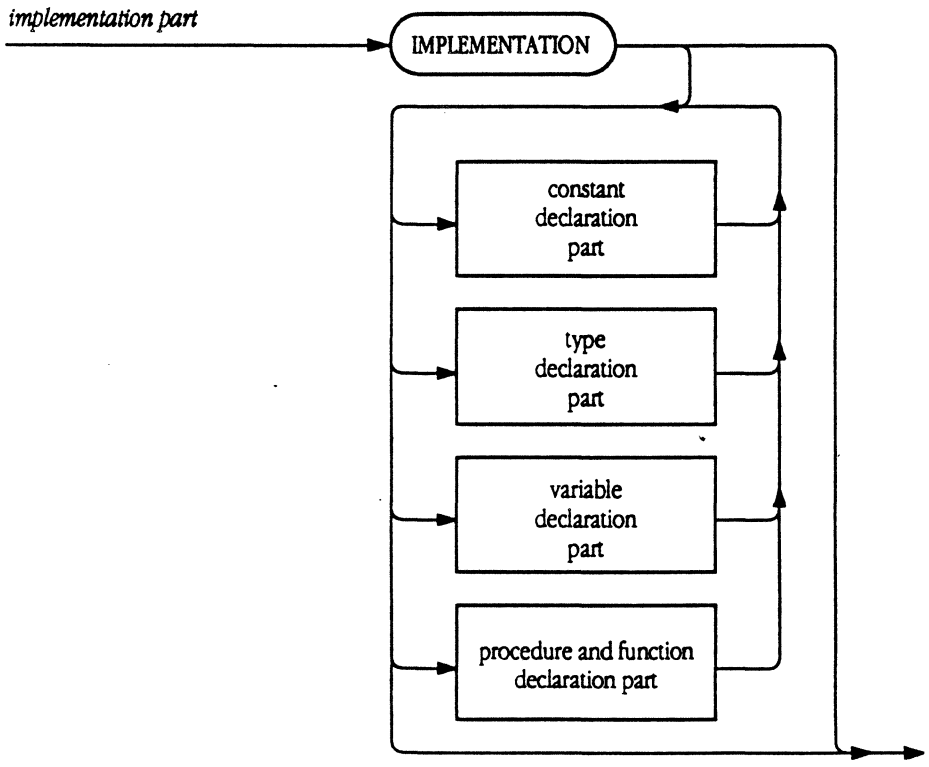
All public procedures, functions, and methods are redeclared in the implementation part. Parameters and function result types can be omitted from these declarations because they were declared in the interface part; the procedure and function blocks, omitted in the interface part, are included in the implementation part. If you repeat parameter lists and function result types, they must be identical to those in the interface part.

In effect, the procedure, function, and method declarations in the interface are like forward declarations, although the FORWARD directive is not used. Therefore, these procedures and functions can be defined and referred to in any sequence in the implementation.

The interface part may contain a USES clause; thus any unit can use another unit.

There is no "initialization" section in MPW Pascal units (unlike Apple II Pascal and Apple III Pascal). If a unit requires initialization of its data, it should define a public procedure that performs the initialization; the host program should then call this procedure.

- *Note:* Global labels cannot be declared in a unit.

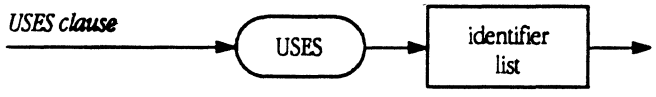


Here is short example of a unit:

```
UNIT Simple;

INTERFACE       {public items declared}
 CONST FirstValue = 1;
 TYPE Num = OBJECT
        val: integer;
        PROCEDURE Bump;
        PROCEDURE Init
       END;
 PROCEDURE AddOne(VAR Incr: integer);
 FUNCTION Add1(Incr: integer): integer;

IMPLEMENTATION
 PROCEDURE AddOne;  {Note lack of parameters...}
  BEGIN
   Incr := Incr+1
  END;
 FUNCTION Add1;     {...and lack of function result type.}
  BEGIN
   Add1 := Incr+1
  END;
 PROCEDURE Num.Bump;
  BEGIN
   val := val+1
  END;
 PROCEDURE Num.Init;
  BEGIN
   val := FirstValue
  END
 END.
```

The USES clause

You write a `USES` clause in a program or unit to access a unit:



The `USES` clause appends the ".p" suffix (which denotes source code) to the unit name and causes the Compiler to open the file. For example, the statement

```
USES QuickDraw, memTypes;
```

opens the files QuickDraw.p and memTypes.p.

The USES clause identifies all units required by the program. These include both units that it uses directly and any other units that are used by those units.

In a host program, the USES clause (if any) immediately follows the program heading. In a host unit, the USES clause (if any) immediately follows the reserved word INTERFACE. Only one USES clause may appear in any host program or unit; it declares all units used by the host program or unit.

See below for the case where a host uses a unit that uses another unit.

It may be necessary to search a particular file for a unit. You can use the $U Compiler command to specify this file, as described in Chapter 13.

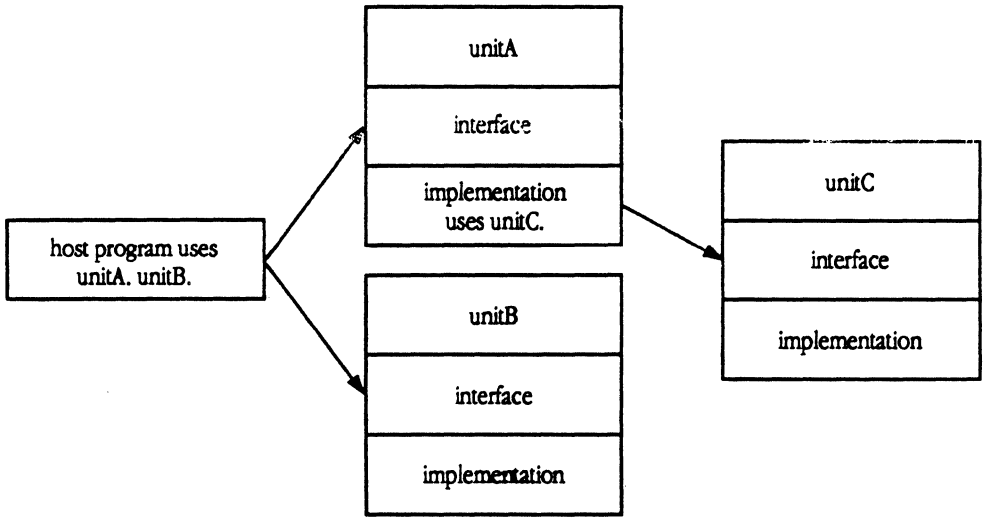
Assume that the example unit is named Simple. The following is a short program that uses Simple. It also uses another unit named Other, which is in file Appl:Other.

```
PROGRAM CallSimple;
 USES {$U APPL:SIMPLE}  {file to search for units}
    Simple,       {use unit Simple}
    {$U APPL:OTHER}  {file to search for units}
    Other;        {use unit Other}
 VAR i: integer;
   n: Num;
 BEGIN
  i := FirstValue;       {FirstValue is from Simple.}
  write('i+1 is ', Add1(i)); {Add1 is defined in Simple.}
  write(xyz(i));        {xyz is defined in Other.}
  New(n);
  n.Init;
  n.Bump;
  write(n.val)
 END.
```

Units that use other units

As explained above, the USES clause in the host program or unit must name all units that are required. Here "required" means that the host directly references something in the interface of the unit. Consider the unit references in Figure 9-1.

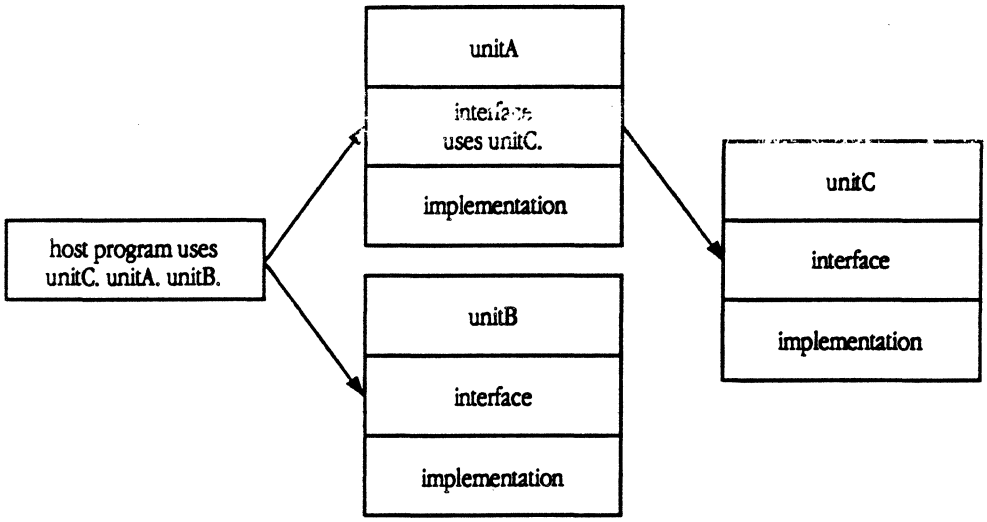
■ **Figure 9-1** Example of simple unit reference



The host program directly references the interfaces of `unitA` and `unitB`; the `USES` clause names both of these units. The implementation part of `unitA` also references the interface of `unitC`, but it is not necessary to name `unitC` in the host program's `USES` clause.

In some cases, the `USES` clause must also name a unit that is not directly referred to by the host. Figure 9-2 is exactly like Figure 9-1 except that this time the interface of `unitA` references the interface of `unitC`, and `unitC` must be named in the host program's `USES` clause. Note that `unitC` must be named before `unitA`.

■ **Figure 9-2** Example of nested unit references



In a case like this, the documentation for `unitA` should state that `unitC` must be named in the `USES` clause before `unitA`.

Automatic symbol table loading

The Pascal Compiler automatically builds a precompiled version of the symbol table for each unit and puts it into the resource fork of the file containing the unit. On subsequent compilations, the Compiler loads this resource instead of compiling the unit. The Compiler does not use the resource if the modification date of the file is later than the date stored when the resource was created or if the values of the compile time options (or compile time variables) that were in effect when the resource was created have changed so as to invalidate the resource. The `-noload`, `-clean`, and `-rebuild` options respectively instruct the Compiler not to create any symbol table resources, to erase all of them, and to rebuild all of them. For more on these options, see Chapter 13 of this manual.

◆ *Note:* If you have units that can't be written to (for instance, on a file server), the `$K` directive (or the `-k` option) can be used to store the symbol resources in a writable directory that you specify. For details on the `$K` directive, see Chapter 13.

Chapter 10 **Files and I/O**

THIS CHAPTER DESCRIBES THE USE OF PASCAL FILES, including the declaration of files in a program. It also includes detailed information on each of the predefined procedures and functions for performing input and output, or **I/O**. ■

Contents

Input/Output routines

MPW Pascal offers you three distinct ways to accomplish input and output in your program:

- by calling the I/O routines in the Macintosh ROM
- by using the I/O procedures and functions that are built into the Pascal Compiler and the library PasLib.o
- by using the I/O procedures in the Integrated Environment library (described in the *Macintosh Programmer's Workshop 3.0 Reference*)

In general, the Macintosh ROM routines provide the most direct way to access the Macintosh screen, keyboard, and mouse. The Pascal built-in routines provide the easiest way to access the contents of files and perform I/O operations with external devices. The Integrated Environment routines are used only by programs that are going to run under the MPW Shell and use its I/O facilities.

Most of what you need to know about accessing the Macintosh Operating System and Toolbox routines is contained in *Inside Macintosh*. Consult the following parts for further information:

- The File Manager chapter tells you how to handle disk files with ROM routines.
- The Event Manager chapter gives you information on how events are handled and on how to use the Event Manager routines to get information from character devices such as the keyboard.
- The QuickDraw chapter contains details of how to provide text output to the screen.
- The Printing Manager chapter discusses printing routines.

The interface files that access the Macintosh ROM routines from MPW Pascal are listed in Appendix E.

The rest of this chapter discusses the I/O routines that come with MPW Pascal. Some of them are built into the Compiler itself; others are included in PasLib. The PasLib procedures and functions have names beginning with `PL`. Any time you use one of them you must include the statement `USES PasLibIntf` in your program or unit. You must also link your program or unit to the library file PasLib.o.

This chapter uses a modified BNF notation instead of syntax diagrams to show the syntax of actual parameter lists for standard procedures and functions. The notation is explained in the Preface.

Pascal files

A Pascal file variable is a structured variable. A file variable resembles an array, in that it consists of a sequence of distinct variable components all of the same type. However, the number of components is indeterminate, and they are not accessed by indexing but by using the predefined I/O procedures and functions.

External files

File variables are used to store data outside of memory, in an **external file**. An external file is either a peripheral device or a named disk file. In order for a program to read or write information using an external file, a file variable must be declared and then associated with the external file.

File variables

The most important feature of a file variable is that its components are not generally in memory, but you can access them as though they were. The components exist outside the program as the contents of an external file.

A file variable is declared along with other variables, using a file type. There are three basic Pascal file types:

- structured files
- text files
- untyped files

The syntax for writing file types is given under "Structured Types" in Chapter 6.

Structured files

A structured file is made up of components called **logical records.** These components may be of any type that is not a file type (or a structured type that contains a file type component at any level of structuring). They do not need to be of type RECORD.

For example, the declarations

```
VAR
 IntVals: FILE OF integer;
 RealVals: FILE OF real;
 CompVals: FILE OF RECORD
             I: integer;
             R: real
           END;
```

create three file variables:

- `IntVals` is a file variable whose logical records are `integer` values.
- `RealVals` is a file variable whose logical records are `real` values.
- `CompVals` is a file variable whose logical records are of `RECORD` type, with an `integer` value and a `real` value in each record.

Text files

The Pascal predefined file type `text` can be used to store any type of data, as long as it is in character format. For example, this declaration creates a file variable of type `text`:

```
VAR NameFile: text;
```

This type of file is most efficient in handling lines of text. It transfers data in blocks between the external device and a buffer, from which your program can access the data efficiently one line or one character or value at a time.

◆ *Note:* There are many cases where this isn't the most efficient way to perform I/O. For example, if you want to store floating-point values in a file, using a file of type `text` would require converting each value to its equivalent in ASCII characters before it is stored in the file. Each time a value is read from the file, it would have to be converted back to its binary representation.

Untyped files

To declare a file variable as untyped use the type identifier `FILE` alone. For example,

```
VAR BlockFile: FILE;
```

Pascal transfers data in and out of such a file without interpreting its internal structure. An untyped file has no file window variable, and it can be used only with the routines `Reset`, `Rewrite`, `Close`, `Eof`, `Blockread`, `Blockwrite`, `Byteread`, and `Bytewrite`. Operations on untyped files are described at the end of this chapter.

Predeclared file variables

MPW Pascal sees all peripheral devices, such as the keyboard and the Macintosh screen, as external files. It predeclares two corresponding file variables of type `text`, called `input` and `output`. Unless specifically redirected, `input` comes from the MPW Shell's **standard input** and `output` goes to the MPW Shell's standard output.

- *Note:* The MPW Shell's diagnostic output file is available when you use the Integrated Environment tools described in the *Macintosh Programmer's Workshop 3.0 Reference.*

At the start of each program execution, the files `input` and `output` are automatically opened for use without being declared. Procedures and functions that can be used with files of type `text` can use them, directly (receiving them as parameters) or indirectly (as the default when the file variable parameter is omitted), without declaring them first. The program should not try to close these files.

- *Note:* The ANSI Standard specifies that `input` and `output` must appear in the program heading if they are used. Many versions of Pascal rely on this. If you are writing code that may be transported to other versions of Pascal, include `input` and `output` in your program heading.

If `input` is used within the program, the standard input file is opened automatically as a read-only file (as though a `Reset` were performed for it) when program execution begins.

If `output` is used within the program, the standard output file is opened automatically as a write-only file (as though a `Rewrite` were performed for it) when program execution begins.

Several of the predefined procedures and functions for use with files of type `text`, described later in this chapter, need not have a file variable explicitly given as a parameter. In these cases, `input` and `output` will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented.

The file window variable

Although a typed file variable may have any number of logical records, only one is accessible at any one time. Each logical record has a number that is its position in the file relative to the first record in the file, which is record number 0. The position of the current record in the file is called the *current file position.* Program access to the current record uses a special variable associated with the file, called a *file window variable.* The file window variable is discussed in Chapter 5.

At any time, there is only one logical record of a file that may be accessed directly through the file window variable. Whenever a file is opened, using any of the procedures described in this chapter, the current file position is set to record 0—the beginning of the file.

The file window variable cannot be used with untyped files.

◆ *Note:* Under certain conditions, such as when the current file position is at the end of the file, the value of the file variable `f` is said to be undefined. It is an error to attempt to use the value of the file window variable `f^` when the value of `f` is undefined. However, assignment to `f^` is still possible if the file may be written to.

Opening a file

Before you can use a file variable, it must be opened. Three procedures are provided for opening existing files and creating and opening new files—`Reset`, `Rewrite`, and `Open`. Each of these procedures is explained in detail later in this chapter.

A new file may be created and opened

- by using `Rewrite`, which creates a new file for write-only, sequential access
- by using `Open`, which creates a new file for read/write, sequential, or random access

An existing file may be opened

- by using `Reset`, which opens the existing file for sequential, read-only access
- by using `Open`, which opens the existing file for read/write, sequential, or random access
- by using `Rewrite`, which opens the existing file for sequential, write-only access

Each of these procedures sets the current file position to zero. You can also use the `Reset` and `Rewrite` procedures to set the current file position of an already-open file to zero. These rules are summarized in Table 10-1.

■ **Table 10-1** File-opening options

Procedure	File kind	Effect
`Reset`	new	An error occurs
`Reset`	existing	Opens an existing file for read-only with sequential access
`Open`	new	Creates a new file for read/write with random access
`Open`	existing	Opens an existing file for read/write with random access
`Rewrite`	new	Creates a new file for write-only with sequential access
`Rewrite`	existing	Opens an existing file for read/write with sequential access; previous contents erased

Closing a file

If you want to associate a file variable with a different external file, you must first close the open file, using the `Close` procedure. This procedure is described in the section "Routines for All Files."

Sequential versus random access

Files may be accessed sequentially or randomly. When a file is opened and accessed sequentially, the first logical record is read or written and then the current file position moves to the numerically next logical record in the file.

Alternatively, files opened with `Open` can be accessed randomly with the `Seek` procedure. The `Seek` procedure takes a parameter whose value is a number referring to the sequence of logical records. By using the `Seek` procedure, you can jump from one record to another, in any order, or access a specific byte position in a `text` file or untyped file.

The function `PLFilepos` may be applied to any file variable; it returns the record number of the current file position. This function is described in detail later in this chapter.

◆ *Note:* The terms *random file* and *sequential file* are commonly used but misleading. Random and sequential are two methods for accessing files—not two kinds of files. Any file can be accessed randomly or sequentially, or both ways. The predefined procedures that support random access are generally used with nontext files, but are not restricted to them.

Routines for all files

The procedures and functions described in this section can be applied to files of all kinds, both typed and untyped.

◆ *Note:* Routines whose identifiers begin with `PL` (such as `PLCrunch`) are defined in the interface file PasLibIntf.p; their code is in the PasLib library.

The Reset procedure

The `Reset` procedure opens an existing file for sequential read-only access or "rewinds" an open file so that its window variable contains the first logical record.

`Reset (`*f* `[, `*filename*`])`

The parameter *f* is a variable reference that refers to a file variable. The parameter *filename* is an optional expression with a string value. If *filename* is given, the file must not already be open. If *filename* is not given, the file must be open.

The value of *filename*, if used, must be a valid Macintosh file pathname, window name, pseudodevice name, or selection specifier.

The statement `Reset (`*f*`)`, when the file specified by *f* is already open, causes the file to be "rewound." The file must have been originally opened with `Open` or `Reset`. If the file was opened with `Open`, it now becomes read-only.

Notice that `Reset` preserves the contents of an existing file, unlike `Rewrite`, which erases the current contents of any file on which it is used.

An error occurs and `IOResult` returns a nonzero value if there is no existing external file with the name specified by *filename*.

The following conditions always hold after `Reset (`*f*`[, `*filename*`])` is executed:

- `Eof (`*f*`)` is `true` if the file is empty. Otherwise, `Eof (`*f*`)` is `false`.
- The current file position is the first logical record of the file (logical record number 0), and the file window variable *f*`^` contains the value of that logical record unless `Eof (`*f*`)` is `true`, in which case the value of *f*`^` is undefined.

The Rewrite procedure

The `Rewrite` procedure creates and opens a new, empty file for write-only access or "rewinds" and erases an open file.

`Rewrite (`*f* `[, `*filename*`])`

The parameter *f* is a file variable. If *filename* is given, the file cannot already be open; if it is, an error occurs. The parameter *filename* is an optional expression with a string value. The string must be a valid Macintosh pathname, window name, pseudodevice name, or selection specifier.

◆ *Note:* `Rewrite (`*f*`)` (with no filename specified), when *f* is not yet open, is not implemented. It is reserved for a future extension of MPW Pascal.

`Rewrite (`*f*`)`, when the file specified by *f* is already open causes the file to be "rewound"; that is, the current file position is reset to the beginning of the file, and any prior contents of the file are deleted. The file must have been opened with `Open` or `Rewrite`. If the file was originally opened with `Open`, it now becomes write-only. `Reset` followed by `Rewrite` causes an error.

`Rewrite (`*f, filename*`)` creates a new external file with the name *filename* and associates the file variable *f* with this external file. If an external file with the name *filename* already exists, it is truncated (that is, the resource fork stays the same although the data fork is deleted).

The following conditions always hold after `Rewrite (`*f*`[, `*filename*`])` is executed:

- `Eof (`*f*`)` is `true`, either because the file is new or because the contents have just been erased.
- The current file position is logical record 0; that is, the first logical record written to the file will become the first logical record of the file. The value of *f*`^` is undefined, and remains undefined until something is written to the file.

The Open procedure

The `Open` procedure opens an existing file or creates and opens a new file for random, read/write access, setting the file window variable to the first logical record. An existing file is not truncated.

`Open (`*f, filename*`)`

The parameter *f* is a variable reference that refers to a file variable. The file may not already be open. The parameter *filename* is an expression with a string value, which must be a valid Macintosh file pathname, window name, pseudodevice name, or selection specifier.

The statement `Open (`*f*`, `*filename*`)` opens an existing external file with the name *filename* and associates the file variable specified by *f* with this external file. If an external file with the name *filename* does not already exist, a new empty file is created. The file is opened for both reading and writing.

The following conditions always hold after `Open (`*f*`, `*filename*`)` is executed:

- `Eof (`*f*`)` is `true` if the file is empty; otherwise, `Eof (`*f*`)` is `false`.
- The current file position is logical record 0, and the file window variable *f*`^` contains the value of that logical record, unless `Eof (`*f*`)` is `true`.

The Close procedure

The `Close` procedure closes an open file. It ends the association between the file variable and the external file, if one exists.

`Close (`*f*`)`

The parameter *f* is a variable reference that refers to a file variable, which must be open.

The statement `Close (`*f*`)` closes *f*. That is, the association between *f* and its external file is broken, and the file system marks the external file closed. All subsequent references to *f* are invalid (except to open it again). In particular, the value of *f*`^` becomes undefined.

If a file has not been closed during program execution, it is closed automatically when the program terminates.

◆ *Note:* Files that have local scope in a procedure or function block are not automatically closed when the procedure or function is exited.

The Eof function

The `Eof` function returns a `boolean` value that indicates whether or not the current file position is the end of the file.

`Eof[(`*f*`)]`

The parameter *f* is a variable reference that refers to a file variable. If *f* is omitted, the function is applied to the predefined file `input`. The file must be open, or an error occurs.

The `Eof` function returns `true` in these cases:

- if the file position is beyond the last logical record of the file
- if the file contains no logical records
- after a `Get` procedure, if the current file position is the last logical record of the file
- after a `Put` procedure, if the logical record written by the `Put` is now the last logical record

In all other cases, `Eof` returns `false`.

The IOResult procedure

```
IOResult
```

The `IOResult` routine returns an `integer` value that indicates the result of the most recently perfomed I/O operation. If `IOResult` returns zero, it means that the last I/O operation was successful. Any nonzero result indicates that the last I/O operation was unsuccessful.

Because files of type `text` are buffered, `IOResult` does not indicate the result of writing to one until the buffer contents are transferred to the external file. A buffer transfer occurs whenever the buffer is full or when there is a call to `PLFlush` or `Close`.

If the `IOResult` code is negative, the number indicates a Toolbox error. The equivalent Macintosh ROM error-return values set in `MacOSErr` are documented in Chapter 4 and in the System Error Handler chapter of *Inside Macintosh.*

If the `IOResult` code is positive, it is an error that has been detected by the Language Library without going to the Toolbox.

The ErrNo variable

The following list documents the values for the variable `ErrNo`. This is a complete list; not all of the ErrNos appear in Pascal.

1 `EPERM` *No permission match*
This error occurs after an attempt to modify a file in some way forbidden except to its creator.

2 ENOENT *No such file or directory*
This error occurs when a file whose filename is specified does not exist or when one of the directories in a pathname does not exist.

3 ENORSRC *Resource not found*
A required resource was not found. This error applies to `faccess` calls that return tab, font, or print record information.

4 EINTR *System service interrupted*
A requested system call cannot be completed. This error may occur if a request to rename a file is unsuccessful.

5 EIO *I/O error*
Some physical I/O error has occurred. This error may in some cases be signaled on a call following the one to which it actually applies.

6 ENXIO *No such device or address*
I/O on a special file refers to a subdevice that does not exist, or the I/O is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.

7 E2BIG *Insufficient space for return argument*
The data to be returned is too large for the space allocated to receive it.

9 EBADF *Bad file number*
Either a file descriptor does not refer to an open file, or a read (or write) request is made to a file that is open only for writing (or reading).

12 ENOMEM *Not enough space*
The system ran out of memory while the library call was executing.

13 EACCES *Permission denied*
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT *Illegal filename*
A filename or volume name was too long or otherwise illegal.

15 ENOTBLK *Block device required*
This error occurs if a non-block file is used when a block device is required.

16 EBUSY *Device or resource busy*
An attempt was made to mount a volume that was already mounted, or to delete a locked file.

17 EEXIST *File exists*
An existing file was mentioned in an inappropriate context.

18 EXDEV *Cross-device link*
This error occurs after a link to a file on another device is attempted.

19 ENODEV *No such device*
An attempt was made to apply an inappropriate system call to a device; for example, read a write-only device.

20 ENOTDIR *Not a directory*
An object that is not a directory was specified where a directory is required, for example, in a path prefix.

21 EISDIR *Is a directory*
An attempt was made to write on a directory.

22 EINVAL *Invalid parameter*
Some invalid parameter was provided to a library function.

23 ENFILE *File table overflow*
The system's table of open files is full, so temporarily a call to open cannot be accepted.

24 EMFILE *Too many open files*
The system cannot allocate memory to record another open file.

25 ENOTTY *Not a typewriter*
This error occurs if the specified file isn't a character file.

26 ETXTBSY *Text file busy*
An attempt was made to open a file that was already open for writing.

27 EFBIG *File too large*
This error occurs if the size of a file was larger than the maximum file size.

28 ENOSPC *No space left on device*
During a write to a file, there is no free space left on the device.

29 ESPIPE *Illegal seek*
An `lseek` was issued incorrectly.

30 EROFS *Read-only file system*
An attempt to modify a file or directory was made on a device mounted for read-only access.

31 EMLINK *Too many links*
An attempt to delete an open file was made.

33 EDOM *Math arg out of domain of func*
This error occurs if the argument of a math function is outside the domain of the function.

34 ERANGE *Math result not representable*
This error occurs when the value of a math function can't be represented within machine precision.

◆ *Note:* Beware of trying to access the value of `IOResult` with an I/O operation, such as

```
Reset(f, 'myfile');
Writeln('IOResult = ', IOResult);
```

In this example, the value of `IOResult` is set by the `Writeln` call, not by the `Reset` call.

▲ **Warning** Some I/O operations may set `IOResult` due to an unseen I/O call. For example, using `Rewrite` to create a new file will return an `IOResult Eof(-39)`, even though the file was rewritten correctly. ▲

The Seek procedure

`Seek(f, n)`

The `Seek` procedure allows you to access any logical record in a file. It does two things:

- It sets the current file position to logical record *n*.
- It reads the new current logical record into the file window variable.

The parameter *f* is a file variable.

The parameter *n* is an expression with a `longint` value that specifies a logical record number in the file. Logical records are numbered from zero. With untyped files or files of the type `text`, *n* is the byte position from the beginning of the file.

For example,

```
Seek(Names, 18)
```

causes the file window variable associated with file `Names` to point to the nineteenth record of the file.

The value of *f*^ becomes the value of that logical record unless *n* is greater than the number of the last logical record of the file, in which case `Eof(f)` becomes `true`, `IOResult` is set to `eofErr(-39)`, and the value of *f*^ becomes undefined. Thus, `Seek(f, maxlongint)` always sets the current file position to the end of file. `EofErr` is not a fatal error, so you can use it to verify that the current file position has actually been moved to the end of the file.

The PLFilepos function

`PLFilepos(`*f*`)`

The `PLFilepos` function returns a value of type `longint` that is the logical record number of the current file position. With untyped files or files of type `text`, it returns the number of bytes from the beginning of the file to the current file position.

The parameter *f* is a variable reference that refers to a file variable. The file must be open.

The PLCrunch procedure

`PLCrunch(`*f*`)`

The `PLCrunch` procedure takes an open file as an argument. It truncates the file at the current position.

The PLPurge procedure

`PLPurge(`*f*`)`

The `PLPurge` procedure deletes the file named by *f*. An error occurs if the file is open.

The PLRename procedure

`PLRename(`*oldname, newname*`)`

The `PLRename` procedure allows you to rename any file.

Oldname and *newname* are strings. *Oldname* is the Macintosh pathname of an existing file, which may be open; *newname* is a new Macintosh pathname. If a file named *newname* already exists, an error occurs and the value of `IOResult` is set to nonzero.

Record-oriented routines

The procedures described in this section are used to access the logical records of a file randomly. Most can be used only with structured files. Get and Put can be used with textfiles. See the section "Using Get and Put with Textfiles" below for details.

The Get procedure

`Get (`*f*`)`

The `Get` procedure does two things:

- It advances the current file position to the next component.
- It reads the new current logical record into the file window variable.

The parameter *f* is a variable reference that refers to a file variable. The file must be open; if it is not, an error occurs.

If a `Get` procedure is performed when no next logical record exists, `Eof (`*f*`)` becomes `true` and the value of *f*`^` becomes undefined.

The Put procedure

`Put (`*f*`)`

The `Put` procedure does two things:

- It writes the file window variable into the file at the current file position.
- It advances the file position to the next logical record.

The parameter *f* is a variable reference that refers to a file variable. The file must be open, and the value of *f*`^` must not be undefined.

The statement `Put (`*f*`)` writes the value of *f*`^` to the external file at the current file position and advances the current file position to the next logical record. If `Eof (`*f*`)` is `true`, `Put (`*f*`)` appends the value of *f*`^` to the end of the file *f* and `Eof (`*f*`)` remains `true`.

The Read procedure with a structured file

The `Read` procedure is usually used with files of type `text`, as descibed below under "Text-Oriented Routines." When used with a structured file, it reads one or more logical records into one or more variables, starting at the current file position and advancing the current position pointer.

`Read(`[*f*`,`] v_1 [`,` v_2`, ...,` v_n]`)`

The optional parameter *f* is the variable of an open structured file. Each parameter v_n must be a variable of a type that is assignment-compatible with the logical records of *f*.

For example, if `Newvals` is a file of type `FILE OF real`, then the procedure call

```
Read(NewVals, Subtotal, Total);
```

requires that `Subtotal` and `Total` be variables of types to which a value of the type `real` can be assigned. If `NewVals` is a variable of type `integer`, for instance, a Compiler error occurs. For the rules that determine possible types in this context, see "Assignment-Compatible Types" in Chapter 4.

The Write procedure with a structured file

The `Write` procedure is usually used with files of type `text`, as described below under "Text-Oriented Routines." When used with a structured file, it writes the value of one or more variables into one or more logical records of the file, starting at the current file position and advancing the current position pointer.

`Write(`[*f*,] p_1 [, p_2, . . . , p_n]`)`

The optional parameter *f* is the variable of an open structured file. Each parameter p_n must be a variable of the type that is assignment-compatible with the logical records of *f*.

For example, if `NewVals` is a file of type `FILE OF real`, then the procedure call

```
Write(NewVals, SubTotal, Total);
```

requires that `SubTotal` and `Total` be variables of types that can be assigned to a value of the type `real` (that is, `real` or `integer`). If `NewVals` is a variable of type `extended`, for instance, a Compiler error occurs. For the rules that determine possible types in this context, see "Assignment-Compatible Types" in Chapter 4.

Text-oriented routines

This section describes input and output routines designed to be used with file variables of type `text`. Text files are distinguished from other kinds of files (for example, `FILE OF char`) by the special significance given to the end-of-line character. This character allows a file of type `text` to be treated as a sequence of lines, rather than a sequence of individual characters. All the text-oriented routines may also be used to read data from the keyboard. An entire line may be read from the file into a string type variable using the `Readln` procedure, and an entire line may be written to the file by using the `Writeln` procedure. You can test for the end-of-line character by using the `Eoln` function described later in this chapter.

- *Note:* When the value of the logical record at the current file position of a file is an end-of-line character, the `Read` and `Readln` procedures read it as a space character (ASCII 32).

The `Read` and `Write` procedures can be applied to any typed file. If used with nontext structured files, they perform as discussed above under "Record-Oriented Routines." However, the procedures `Readln` and `Writeln` depend upon the presence of the `Eoln` character, which appears only in files of type `text`.

- *Note:* None of the predefined procedures and functions in this section need an explicit file variable parameter. If no file is named, one of the predefined files, `input` or `output`, will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented. Remember that `input` and `output` are predeclared as files of type `text`.

The Read procedure

`Read([` *f*`,] ` v_1 `[,` v_2`, ..., ` v_n`])`

The `Read` procedure reads one or more values from a text file into one or more variables.

If *f* is given, it must be a variable reference that refers to a file variable of type `text`. The file must be open. If *f* is omitted, it is assumed to be the predefined text file `input`.

Each v_n is a variable reference that refers to a variable of one of the following types:

- `char` or a subrange of type `char`
- `integer`, `longint`, or a subrange of `integer` or `longint`
- one of the real types
- a scalar type (including `boolean`) or a subrange of a scalar type
- a string type
- a `PACKED ARRAY OF char`

`Read` with an array element follows the rules for the element's type. The other possibilities are discussed below.

Read with a char variable

A `Read` performed with a `char` type variable is considered equivalent to this compound statement:

```
BEGIN
  v := ff^;
  Get(ff)
END
```

In this example, `v` is a variable of type `char` and `ff` is a `FILE OF char`. Remember that if the current file position is at an end-of-line character, `ff^` contains a space character.

Read with an integer variable

A `Read` procedure performed with an `integer` or `integer` subrange variable reads a sequence of characters that form a signed, whole number in the range of type `integer` or type `longint`. If the sequence of characters is a valid representation of an `integer`, the `integer` value is assigned to the variable. Otherwise, an error occurs.

When an `integer` is being read, the sequence of characters, spaces, tabs, and end-of-line characters preceding the first digit or the sign is skipped. Reading ceases as soon as a character is reached that (together with the characters already read) does not form part of a signed whole number; or when `Eof(`*f*`)` becomes `true`.

If a signed whole number is not found after skipping any preceding spaces, tabs, and end-of-line characters, an errors occurs and `IOResult` returns a value of –1025.

The following conditions are true immediately after a `Read` from a text file with an `integer` variable:

- The current file position is the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- `Eof(`*f*`)` will return `true` if the last character in the numeric string was the last character in the file.
- `Eoln(`*f*`)` will return `true` if the last character in the numeric string was the last character on the line.

Read with a real variable

A `Read` performed on a text file with a variable of one of the real types reads a sequence of characters that forms a signed number. If the sequence of characters is a valid representation of a value of a real type, the value is assigned to the variable. Otherwise, an error occurs.

When a value of type `real` is being read, any sequence of blanks and tabs preceding the first digit or the sign is skipped. Reading ceases as soon as a character is reached that (together with the characters already read) does not form part of a signed number, or when `Eof(`*f*`)` becomes `true`.

If a signed real number is not found after skipping any preceding spaces and tabs, NaN is returned and the `Invalid` exception, described in Appendix G, is signaled.

◆ *Note:* In addition to standard Pascal syntax, MPW Pascal regards, for example, `inf`, `NaN`, `NaN(0)`, `NaN( )`, `NaN(39)`, `.369`, `.369312`, `1.`, and `1.e9` all as valid and real numbers.

Immediately after a `Read` from a text file with a variable of the real type, the following conditions are true:

- The current file position is the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- `Eof(`*f*`)` will return `true` if the last character in the numeric string was the last character in the file.
- `Eoln(`*f*`)` will return `true` if the last character in the numeric string was the last character on the line.

Read with a string variable

A `Read` performed with a string variable reads a sequence of characters up to, but not including, the next end-of-line character, or until the end of the file. The resulting character string is assigned to the variable. It is an error if the number of characters read exceeds the size attribute of the variable.

◆ *Note:* With a string variable, `Read` does not skip to the next line after reading; an end-of-line character is left in the file buffer. For this reason, you cannot use successive `Read` calls to read a sequence of strings; after the first `Read`, each subsequent `Read` will access the end of line (instead of a character) and will read a zero-length string. Instead, you must use `Readln` to read string values; `Readln` skips to the beginning of the next line after each input.

The following conditions are true immediately after a `Read` from a text file with a string variable:

- The current file position is the character following the last character in the string, unless the last character in the string was the last character in the file.
- `Eof(`*f*`)` will return `true` if the last character in the string was the last character in the file.
- `Eoln(`*f*`)` will return `true` unless `Eof(`*f*`)` is `true`, in which case `Eoln(`*f*`)` is undefined.

The Readln procedure

`Readln(`[*f*`,`] *v*$_1$ [`,` *v*$_2$`, ..., `*v*$_n$]`)`

The `Readln` procedure is an extension of `Read`. It reads a sequence of characters until the next character is the end-of-line character. It then skips to the beginning of the next line in the input file. Because `Readln` depends on finding the end-of-line character, it can be used only with files of type `text`.

The parameters allowed with `Readln` are the same as those for `Read`. In addition, you can use `Readln`

- with no input variables
- with no parameters

If the first parameter does not specify a file or if no parameters are used, `Readln` reads from the standard file `input`.

When `Readln` is used without input variables, it advances the current file position to the beginning of the next line, if there is one. If there is no next line, it advances the current file position to the end of the file.

The following conditions are true immediately after a `Readln`, regardless of the type of any input variable used:

- `Eof(`*f*`)` will return `true` if the line read was the last line in the file.
- `Eoln(`*f*`)` will return `false` unless the line following the line read is empty.

The Write procedure

`Write(`[*f*,] p_1 [,p_2, ...,p_n]`)`

The `Write` procedure writes one or more values to a text file. The parameter *f* (if given) is a variable reference that refers to a file variable of type `text`. The file must be open. If *f* is omitted, the procedure writes to the predefined file `output`.

Each p_n parameter is a `Write` parameter. At least one `Write` parameter must be present. The value of each `Write` parameter, p_n, is given by an *output expression*, which may be of type `char`, `integer`, `real`, `STRING`, `PACKED ARRAY OF char`, or `boolean`.

Besides complex expressions, such output expressions also include single variables and constants. The effects of using the `Write` procedure with these different types are discussed below.

Each `Write` parameter has the form

OutExp `[:` *MinWidth* `[ :` *DecPlaces* `]]`

where *OutExp* is an output expression. *MinWidth* and *DecPlaces* are optional expressions with `integer` values. For example, in the statements

```
Write(NewVals, Total);
Write(NewVals, Total:8:4);
```

`Total` is the output expression. The value of the real type variable `Total` is to be written to file `NewVals`. In the first case, the optional *MinWidth* and *DecPlaces* specifications are omitted. In the second case, *MinWidth* is eight and *DecPlaces* four. `Total` will be written to a field eight spaces wide and with four characters to the right of the decimal place.

MinWidth specifies the minimum field width. *MinWidth* must be greater than or equal to zero. Exactly *MinWidth* characters are written (using leading spaces if necessary), except when *OutExpr* has a value that must be represented in more than *MinWidth* characters, in which case the exact number of characters needed is written. *MinWidth* can be used with an *OutExp* of any type that is valid in a `Write` parameter.

DecPlaces specifies the number of decimal places to be used in the fixed-point representation of a real value. It can be specified only if *OutExpr* has a real type value and if *MinWidth* is also specified. If specified, it must be greater than zero. If *DecPlaces* is not specified and the value is one of the real types, a floating-point representation is written. Floating-point representation is discussed later in this chapter.

Write with a char value

If `Write` is given a variable of type `char` and *MinWidth* is not specified, the character value of *OutExpr* is written to the specified file. If *MinWidth* is included, exactly *MinWidth* –1 spaces are written, followed by the character value of *OutExpr*. For example,

```
Write(Names, Initial);
Write(Names, Initial:3);
```

In the first example, the character that is the value of `Initial` is written to the file, without leading spaces. In the second example, `Initial` is written, preceded by two spaces.

Write with an integer value

If *OutExpr* is of type `integer`, its decimal representation is written to the specified file as if by this algorithm:

```
BEGIN
  FOR I := 1 TO MinWidth - (length(OutDigits) + 1) DO
    Write(ff, ' ');
  IF OutExpr < 0 THEN
    Write(ff, '-')
  ELSE
    Write(ff, ' ');
  Write(ff, OutDigits)
END;
```

The parameter `ff` represents the variable referenced by *f*. OutDigits is a string value that contains the decimal representation of the absolute value of *OutExpr*, with no leading zeros unless the value of *OutExpr* is zero, in which case `OutDigits` contains the single character "0".

For example, if the decimal representation of the value of *OutExpr* is 4545 and `MinWidth` is given as eight, the `FOR` statement will write three space characters to a file. Because `OutExpr` is not less than zero, the `IF` clause will not execute and the `ELSE` clause will output one more space. Finally, the last `Write` statement will output the decimal number 4545 to the file.

Here are some additional syntax rules for using `Write` with an `integer` value:

- If *MinWidth* is used and its value is greater than the number of digits in the decimal representation of the value to be written, leading spaces will be written to the left of the number. The number of spaces depends upon the *MinWidth* specification.
- If the value of *OutExpr* is less than zero, a minus sign (ASCII $2D) is written to the file, denoting a negative value.
- If *MinWidth* is omitted, it is given a default value of eight.

Write with a value of type real

If *OutExpr* has a real value, its decimal representation is written to the specified file. This representation depends upon the value of the parameter *DecPlaces* (if it is present).

If *DecPlaces* is present, a fixed-point representation is written. If *DecPlaces* is absent, a floating-point representation is written. These two cases are discussed separately below.

Fixed-point representation

Assume that `IntDigits` is a string value containing the decimal representation of this expression:

```
Trunc(Abs(OutExpr))
```

`IntDigits` contains no leading zeros (unless the value of `OutExpr` is zero, in which case `IntDigits` contains the single character 0). This expression is the value for the portion of `OutExpr` to the left of the decimal point.

Now assume that `FracDigits` is a string value that contains the decimal representation of this expression:

```
Round((Abs(OutExpr) - Trunc(Abs(OutExpr))) * 10^DecPlaces)
```

with enough leading zeros to make `Length(FracDigits)` equal to `DecPlaces`.

Then the fixed-point representation is written to the file using this algorithm:

```
BEGIN
 IF MinWidth >= length(IntDigits)+length(FracDigits)+2 THEN
   Write(ff, ' ': MinWidth-TotalDigits-3);
 IF OutExpr < 0 THEN Write(ff, '-')
   ELSE
     IF MinWidth >= length(IntDigits)+length(FracDigits) +2 THEN
       Write(ff, ' ');
       Write(ff, IntDigits, '.', FracDigits)
END;
```

If `MinWidth` is omitted from the `Write` parameter, it is assumed to be ten.

Floating-point representation

The algorithm used to write a floating-point representation works in this way. The expression `Abs(OutExpr)` can be represented in floating-point notation in this form:

m.n $*10^{e}$

In this expression, *m* is always a digit from one to nine, unless the value of `OutExpr` is zero. Assume that `IntDigit` is a string value that contains the decimal representation of *m*—a single digit. Assume that `FracDigits` is a string value that contains the first `MinWidth` –9 digits of the decimal representation of *n* rounded, or with trailing blanks retained and trailing zeros added if necessary. Assume that `ExpDigits` is a string value that contains the decimal representation of `Abs(e)` with enough leading blanks to make `Length(ExpDigits)` equal to four. Also assume that `NegExp` has the value `true` if `e` <0 and otherwise is `false`. Given these assumptions, the following is the algorithm for writing a floating-point representation:

```
BEGIN
  IF OutExpr < 0 THEN Write(ff, '-') ELSE Write(ff, ' ');
  Write(ff, IntDigit, '.', FracDigits, 'E');
  IF NegExp THEN Write(ff, '-') ELSE Write(ff, '+');
  Write(ff, ExpDigits)
END;
```

Write with a string value

The results of using `Write` with a string variable depend upon the length attribute of the string that appears as the *OutExpr* and whether or not *MinWidth* is specified.

Here are the rules:

- If *MinWidth* is specified and the length of the string is less than *MinWidth*, then the string is written preceded by a number of spaces equal to *MinWidth* minus the length of the string.
- If *MinWidth* is specified and the length of the string is greater than *MinWidth*, then the first *MinWidth* number of characters are written.
- If *MinWidth* is specified and the length of the string equals *MinWidth*, or if *MinWidth* is not specified, the entire string value is written on the file.

For example, in the statement

```
Write(LastName: 8);
```

`LastName` is a string variable with a size of ten. If `LastName` holds a value that is either nine or ten characters, only eight will be written to `output`.

Write with a packed array of char

If *OutExpr* is a `PACKED ARRAY OF char`, the effect is the same as writing a string whose length is the number of logical records in the type.

Write with a boolean value

If the value of *OutExpr* is type `boolean`, the string `' TRUE'` (with a leading space) or the string `'FALSE'` is written to the file *f*. The default value of *MinWidth* is five. If *MinWidth* is greater than five, leading spaces are added; if *MinWidth* is less than five, the character `T` or `F` is written, padded with spaces as if a value of type `STRING[1]`.

The Writeln procedure

`Writeln([` *f*`,`] *p*$_1$ `[,` *p*$_2$`, ..., ` *p*$_n$`])`

The `Writeln` procedure is an extension of `Write`. It performs the same actions and then writes an end-of-line character to the output file.

The parameters are the same as those used with `Write`, except the `Write` parameters can be entirely omitted; without them, `Writeln` writes an end-of-line character to the output.

The Eoln function

`Eoln[(`*f*`)]`

The parameter *f* is a variable reference that refers to a file variable of type `tex`. The file must be open. If *f* is omitted, the function is applied to the predefined file `input`.

`Eoln` returns `true` if the character at the current file position is an end-of-line character. An error occurs if `Eoln (` *f* `)` is applied to a nontext file or if *f* is write-only. If `Eof (` *f* `)` is `true`, `Eoln ( )` is undefined.

The Page procedure

`Page[(`*f*`)]`

The `Page` procedure sends a form feed character (ASCII 12) to the file designated by *f*. If the file parameter is omitted, the character is sent to the predefined file `output`.

The PLSetVBuf procedure

`PLSetVBuf` (*f, bufptr, style, bufsize*)

The `PLSetVBuf` procedure allows you to specify your own buffer for use with files of type `text`.

The parameter *f* is a variable reference that refers to a file variable of type `text`. The parameter *bufptr* is a pointer to a `PACKED ARRAY OF char` to be used as a text I/O buffer. *Bufsize* is an `integer` that gives the size of the buffer in bytes. *Style* is an `integer` that determines how buffering is performed.

Style	Effect
`_IOFBF`	I/O is file buffered
`_IOLBF`	Output is line buffered; the buffer is flushed when full or at `Eoln`
`_IONBF`	I/O is unbuffered; *bufptr* and *bufsize* are ignored

The system normally allocates a file's buffer when the first read or write operation is performed on it. To allocate your own buffer, call `PLSetVBuf` after the file is opened but before the first read or write operation. If the value of *bufptr* is `NIL`, the system allocates a buffer of size *bufsize* at the first read or write operation. Be sure to close the file before deallocating its buffer.

The PLFlush procedure

`PLFlush(`*f*`)`

The `PLFlush` procedure causes the contents of the current output buffer associated with the file *f* of type `text` to be written to the file.

The Get and Put procedures with text files

The `Get` and `Put` procedures can be used for character-at-a-time I/O. `Get` with a text file differs from `Get` with a structured file only in that a character is not read until a program reads the file window variable. This behavior of `Get` makes it possible to interact with infromation entered from the keyboard.

△ **Important** Don't mix `Get` and `Put` with `Read` and `Write`. Currently they're not compatible. △

Routines for untyped files

The following routines can be used only on untyped files—that is, variables of type `FILE` with no specified logical record type. With `Blockread` and `Blockwrite`, an untyped file is treated as a sequence of 512-byte blocks. With `Byteread` and `Bytewrite`, it is treated as a sequence of bytes. In both cases, the file bytes are not type-checked but are considered as raw data. This can be useful for applications where the data need not be interpreted at all during I/O operations.

The blocks in an untyped file are considered to be numbered sequentially starting with logical record 0. The system keeps track of the current block number; it is block 0 immediately after the file is opened. Each time a block is read, the current block number is incremented. By default, each I/O operation begins at the current block number; however, an arbitrary block number can be specified.

An untyped file has no file window variable, and it cannot be used with the `Get` or `Put` procedure or with any of the text-oriented I/O procedures. It can only be used with `Reset`, `Rewrite`, `Open`, `Close`, `Seek`, `Eof`, and the four functions described below.

To use untyped file I/O, an untyped file is opened with `Open`, `Reset`, or `Rewrite`; the `Blockread`, `Blockwrite`, `Byteread`, and `Bytewrite` functions may then be used for input and output.

The Blockread function

`Blockread`(*f, databuf, count* [, *blocknum*])

The `Blockread` function reads one or more 512-byte blocks of data from an untyped file to a program variable and returns an `integer` representing the number of blocks read. Its parameters are the following:

- The parameter *f* is a variable reference that refers to a variable of type `FILE`. The file must be open.
- *Databuf* is a variable reference that refers to the variable into which the blocks of data will be read. The size and type of this variable are not checked; if it is not large enough to hold the data, other data may be overwritten and the results are unpredictable.
- *Count* is an expression with an `integer` value. It specifies the maximum number of blocks to be transferred. `Blockread` will read blocks until this limit is reached, the end-of-file is reached, or an error occurs.

- *Blocknum* is an optional expression with an `integer` value. It specifies the starting block number for the transfer. If it is omitted, the transfer begins with the current block. Thus the transfers are sequential if the *blocknum* parameter is never used; if a *blocknum* parameter is used, it provides random access to blocks.

After the last block in the file has been read, the current block number is unspecified and `Eof` (*f*) is `true`. Otherwise, `Eof` (*f*) is `false` and the current block number is advanced to the block after the last block that was read. If `Blockread` reads fewer than *blocknum* blocks, it returns the actual number of blocks read. If the end of the file occurs while the last block is being read, the remainder of the block is filled with zero bytes. If `Eof` (*f*) is `true` when `Blockread` is called, `Blockread` returns zero and `IOResult` returns a value of `eofErr(-39)`.

The Blockwrite function

`Blockwrite`(*f, databuf, count*[, *blocknum*])

The `Blockwrite` function writes one or more 512-byte blocks of data from a buffer to an untyped file and returns an `integer` representing the number of blocks written.

- The parameter *f* is a variable reference that refers to a variable of type `FILE`. The file must be open.
- *Databuf* is a variable reference that refers to the variable from which the blocks of data will be written. The size and type of this variable are not checked.
- *Count* is an expression with an `integer` value. It specifies the maximum number of blocks to be transferred. `Blockwrite` will write blocks up to this limit unless an error occurs.
- *Blocknum* an optional expression with an `integer` value. It specifies the starting block number for the transfer. If it is omitted, the transfer begins with the current block. Thus the transfers are sequential if the *blocknum* parameter is never used; if a *blocknum* parameter is used, it provides random access to blocks.

If disk space runs out during data transfer, the current block number is unspecified. `Blockwrite` returns the actual number of blocks written and sets `IOResult` to a nonzero value. Otherwise, the current block number is advanced to the block after the last block that was written.

The Byteread and Bytewrite functions

`Byteread(` *f, databuf, count* [, *blocknum*])

`Bytewrite(` *f, databuf, count* [, *blocknum*])

The `Byteread` and `Bytewrite` functions perform identically to `Blockread` and `Blockwrite`, with four differences:

- They transfer bytes of data instead of blocks.
- The type of the parameters *bytenum* and *count* in the function result is `longint` instead of `integer`.
- The value *bytenum* is the current byte position in the file. The first byte is numbered zero.
- `Byteread` and `Bytewrite` return the number of bytes transferred.

◆ *Note:* Mixing block and byte untyped file functions can result in confusion unless their *blocknum* and *bytenum* parameters are used to adjust the current file position. A block function always transfers the next 512 bytes; after a byte function, this may no longer conform to a natural block boundary.

Chapter 11 **Predefined Routines**

THIS CHAPTER DESCRIBES ALL THE PREDEFINED ("BUILT-IN") PROCEDURES and functions in MPW Pascal, except for the I/O procedures and functions described in Chapter 10. The routines described in Appendix G are contained in the SANE libraries, rather than being implemented by the MPW Pascal Compiler or the PasLib library.

The Macintosh also has more than 500 ROM routines available, which are described in *Inside Macintosh.* Those routines ease implementation of the Macintosh user interface and provide program services.

Standard procedures and functions are predeclared. Predeclared entities act as if they were declared in a block surrounding the program, so no conflict arises from a declaration that redefines the same identifier within the program.

◆ *Note:* Predefined procedures and functions cannot be used as actual parameters for procedures and functions.

This chapter uses a modified BNF notation instead of syntax diagrams to indicate the syntax of actual parameter lists for standard procedures and functions. The notation is explained in the Preface. ■

Contents

Exit and halt procedures

Two procedures, `Exit` and `Halt`, let you terminate current program execution unconditionally.

The Exit procedure

The `Exit` procedure exits immediately from a specified procedure or function or from the main program.

```
Exit({id | PROGRAM})
```

The parameter *id* is the identifier of a procedure or function, or of the main program, in the scope of the `Exit` call. You can also use the reserved word `PROGRAM` to identify the currently executing program.

The call `Exit (`*id*`)` causes an immediate exit from *id*. Essentially, it causes a jump to the end of *id*. The routine identified by *id* must be part of the current dynamic calling chain.

`Exit(PROGRAM)` sets the MPW `{status}` variable to zero.

The Halt procedure

```
Halt
```

`Halt` (with no parameters) causes an immediate exit from the main program and sets the MPW `{status}` variable to one.

Dynamic allocation procedures

These procedures are used to manage the heap, a memory area within the application heap zone. (See the Memory Manager chapter of *Inside Macintosh* for details of memory allocation on the Macintosh.) The `PLHEAPINIT` procedure lets you specify the size of the heap you wish to use with your program. The `PLSetNonCont`, `PLSetMErrProc`, and `PLSetHeapType` procedures let you control the characteristics of the heap. The procedure `New` is used for all allocation of heap space by the program. The `Mark` and `Release` procedures are used together to deallocate all of a marked part of heap space. The `Dispose` procedure is used to deallocate a single identified variable. The `Heapresult` function is used to return the status of the preceding dynamic allocation operation.

◆ *Note:* The routines whose names begin with `PL` are located in PasLibIntf.p.

Except when dealing with object types, the `NewHandle` procedure in the Macintosh ROM can also be used to allocate heap space. `NewHandle` returns handles (double indirect pointers) rather than ordinary pointers. The Memory Manager can then maintain heap space for you, compacting the heap when necessary and allocating more efficiently use of memory space. See *Inside Macintosh* for details of `NewHandle`. When creating new objects, always use the `New` procedure described here; it calls `NewHandle` with the proper arguments. Objects are always relocatable.

Except when creating objects, the allocation procedures described here allocate nonrelocatable blocks in the heap. The Memory Manager cannot move those blocks in order to free larger contiguous blocks for later allocation. The `New` procedure has the advantage that it lets you allocate heap space without having to specify its size.

The PLHeapInit procedure

```
PLHeapInit(sizeHeap: longint; heapDelta: longint;
      memerrProc: UNIV PascalPointer, allowNonCont: boolean, forDispose:boolean)
```

The `PLHeapInit` procedure initializes the heap, using information you supply to determine the characteristics of the heap.

The *sizeHeap* parameter takes a `longint` value that represents the size of the heap. MPW Pascal's built-in heap-initialization routine automatically allocates 5000 bytes of heap space. Using `PLHEAPINIT`, you can specify a heap size other than 5000 bytes.

The *heapDelta* parameter specifies the size in bytes of additional space to be added to the heap if *allowNonCont* is `true`. `New` allocations may not be adjacent to the existing heap.

The *memerrProc* argument is a procedure pointer that enables you to specify a routine to be executed if a memory error, such as heap overflow, occurs.

The *allowNonCont* parameter is a `boolean` value. If it is set to `true`, additional heap space equal to the argument given to *heapDelta* will be allocated when the initial heap space is exhausted. If it is set to `false`, `PLHeapInit` will ignore *heapDelta*, and no aditional space is allocated.

The *forDispose* parameter is also a `boolean` value. The *forDispose* parameter must be set to *true* if you want to use the `Dispose` procedure. Otherwise, an error occurs if you attempt to call `Dispose`. The default setting is `false`.

`PLHeapInit` should be called by your main program.

The PLSetHeapCheck procedure

`PLSetHeapCheck (`*DoIt:* `boolean)`

Whenever heap space is allocated or deallocated, a consistency check is normally performed on the heap. The procedure `PLSetHeapCheck` allows you to suspend this checking process by setting the `boolean` parameter *DoIt* to `false`. It remains suspended until a subsequent `PLSetHeapCheck` procedure is made with *DoIt* `true`.

`PLSetHeapCheck` should be called by your main program.

The PLSetNonCont procedure

`PLSetNonCont (`*allowNonCont:* `boolean)`

The `PLSetNonCont` procedure lets you set the additional heap space flag without calling `PLHeapInit`. The *allowNonCont* parameter is a `boolean` value. If it's set to `true`, additional heap space will be allocated if the current heap is full and cannot be extended.

The PLSetMErrProc procedure

`PLSetMErrProc (`*memerrProc:* `univ PascalPointer)`

The `PLSetMErrProc` procedure allows you to specify a procedure to be executed in case of a memory error. The parameter *memerrProc* points to the procedure.

The PLSetHeapType procedure

`PLSetHeapType (`*forDispose:* `boolean)`

The `PLSetHeapType` procedure lets you specify, without accessing `PLHeapInit`, whether or not use of the `Dispose` procedure is to be allowed in your program. If the `boolean` *forDispose* is `true`, `Dispose` is allowed.

- *Note:* Be careful if you change the heap type in the middle of the program. Pointers allocated for one type of heap are not compatible with pointers allocated for the other type of heap.

The New procedure

New (*p*[, t_1, ..., t_n])

The New procedure allocates a new dynamic variable and sets a pointer variable to point to it.

The parameter *p* is a variable reference that refers to a variable of any pointer type. It may also be an object type reference variable, in which case New creates a new object of that type. The parameter *p* is a variable parameter; it can be a pointer variable or object type reference variable of any type.

The optional parameters t_1, ..., t_n are constants, used only when allocating a variable of record type with variants (see below).

If *p* is a pointer variable, New (*p*) allocates a new variable of the base type of *p* and makes *p* point to it. The variable can be referenced as *p*^.

◆ *Note:* The New procedure is not the same as the NewPtr function described in *Inside Macintosh*. When you call NewPtr, you give a size value and the result does not have a type. The result of New always points to an identified variable of a specific type.

If *p* is an object type reference variable, space is allocated for an object of the variable's type and a handle (a double indirect pointer) is assigned to *p*. You do not, however, use pointer symbols to reference values that are in fields of the new object. You reference fields of objects as if they were fields of ordinary records. Successive calls do not necessarily allocate contiguous areas. In general, objects can move when the heap is compacted.

If the heap does not contain enough free space to allocate the new variable, *p* is set to NIL and a subsequent call to the Heapresult function will return a nonzero result.

If the base type of *p* is a record type with variants, New (*p*) allocates enough space to allow for the largest variant. The form

New(*p*, t_1, ..., t_n)

allocates a variable with space for the variants specified by the tag values t_1, ..., t_n (instead of enough space for the largest variants). The tag values must be constants; they must be listed contiguously and in the order of their declaration. The tag values are not assigned to the tag fields by this procedure.

Trailing tag values can be omitted. The space allocated allows for the largest variants for all tag values that are not specified.

▲ **Warning** When a record variable is dynamically allocated with explicit tag values as shown above, you should not make assignments to any fields of variants that are not selected by the tag values. You should also not assign an entire record to this record. If you do either of these things, other data can be overwritten without any error being detected at compile time. ▲

The Dispose procedure

`Dispose(`*p*`)`

The `Dispose` procedure deallocates an identified variable or an object.

The parameter *p* is a variable reference that refers to a variable of any pointer type or object type reference variable. It is a variable parameter.

`Dispose` releases the space allocated to a dynamic variable or object. It is an error if *p* is undefined or `NIL`. After `Dispose` executes, the value of *p* is undefined. All other references to the identified variable or object that was reached through *p* are also undefined.

◆ *Note:* You must use the `PLHeapInit` or `PLSetHeapType` procedure, described above, to set the *allowDispose* flag to `true` before you use the `Dispose` procedure. In addition, the *allowDispose* flag must have been `true` at the time that *p* was established by a call to `New`.

The Heapresult function

`Heapresult`

The `Heapresult` function returns an `integer` representing the status of the most recent dynamic allocation operation.

The `Heapresult` function returns an `integer` code that reflects the status of the most recent call on `New`, `Mark`, `Release`, `Memavail`, or `PLHeapInit`. The codes are given below:

Code	Meaning
0	Successful operation
-1051	Illegal size request (larger than total heap space)
-1052	Invalid pointer
-1053	Insufficient room in heap

The Mark procedure

`Mark(`*p*`)`

The `Mark` procedure sets a pointer to a heap area.

The parameter *p* is a variable reference that refers to a variable of any pointer type. It is a variable parameter.

`Mark`(*p*) causes the pointer *p* to point to the start of the current free area in the heap. The pointer *p* is also placed on a stacklike list for subsequent use with the `Release` procedure (see below).

The Release procedure

`Release(`*p*`)`

The `Release` procedure deallocates all variables in a marked heap area.

The parameter *p* is a variable reference that refers to a pointer variable. It must be a pointer that was previously set with the `Mark` procedure.

`Release`(*p*) deallocates all areas allocated since the pointer *p* was passed to the `Mark` procedure.

The Memavail function

`Memavail`

The `Memavail` function returns a `longint` that gives the maximum possible amount of available heap space. It has no parameters.

`Memavail` returns the maximum number of words (not bytes) of heap space that can currently be made available to the `New` procedure (assuming that the Pascal heap is allowed to grow in size). Note that the result of `Memavail` can change over time even if the program does not allocate any heap space, because of other memory management activities.

Transfer functions

Transfer functions transfer a value from an expression of one type to an expression of another type. See "Real Types" in Chapter 4 for a discussion of real and extended values.

The Trunc function

`Trunc (`*x*`)`

The `Trunc` function converts an extended value to a `longint` value. Its parameter *x* is an expression with a value of type `extended`. `Trunc (`*x*`)` returns a `longint` result that is the value of *x* rounded to the largest whole number between zero and *x* (inclusive).

The Round function

`Round (`*x*`)`

The `Round` function converts an extended value to a `longint` value. Its parameter *x* is an expression with a value of type `extended`. If *x* is exactly halfway between two whole numbers, the result is the whole number with the greatest absolute magnitude.

The Ord4 function

`Ord4 (`*x*`)`

The `Ord4` function converts a scalar type or pointer type value to type `longint`. Its parameter *x* is an expression with a value of scalar type or pointer type. `Ord4 (`*x*`)` returns the value of *x*, converted to type `longint`.

If *x* is of type `longint`, the result is the same as *x*.

If *x* is of pointer type, the result is the corresponding physical address of type `longint`.

If *x* is of type `integer`, the result is the same numerical value represented by *x* but of type `longint`. This is useful in arithmetic expressions. For example, consider the expression

```
abc*xyz
```

where both `abc` and `xyz` are of type `integer`. By the rules given in Chapter 3, the result of this multiplication is of type `integer` (16 bits). If the mathematical product of `abc` and `xyz` cannot be represented in 16 bits, the result is the low-order 16 bits. To avoid this, the expression can be written as

```
Ord4(abc)*xyz
```

This expression causes 32-bit arithmetic to be used, and the result is a 32-bit `longint` value.

If *x* is of a scalar type other than `integer` or `longint`, the numerical value of the result is the ordinal number determined by mapping the values of the type onto consecutive nonnegative integers starting at zero.

The Pointer function

```
Pointer(x)
```

The `Pointer` function converts an `integer` or `longint` value to pointer type. Its parameter *x* is an expression with a value of type `integer` or `longint`. `Pointer(x)` returns a pointer value that corresponds to the physical address *x*. This pointer is of the same type as `NIL` and is assignment compatible with any pointer type. The value of `Pointer(0)` is `NIL`.

Arithmetic functions

The MPW Pascal arithmetic functions that take parameters of real types reside in the Macintosh ROM and/or the 68881. The Pascal Compiler generates the code necessary to call them from Pascal source text. For information about the limits and accuracy of these functions, consult the *Apple Numerics Manual*. For more information on the 68881 functions, consult Motorola's *MC68881 Floating-Point Coprocessor User's Manual*. In general, any result returned by an arithmetic function is an approximation, although the result of the `Abs` function is exact.

Functions that do not have parameters of real types are implemented by code generated by the Compiler.

In this section, a numeric value is defined as an expression involving constants and variables of types `extended`, `double`, `real`, `comp`, `longint`, or `integer`. Numeric values are therefore of type `extended`, `longint`, or `integer`.

When you set the `-MC68881` option, the Compiler generates direct calls to the 68881 for several of the functions described below. See Appendix G for details.

The Odd function

`Odd(`*x*`)`

The `Odd` function tests whether a whole-number value is odd, returning a `boolean` value. Its parameter *x* is an expression with a value of type `integer` or `longint`. `Odd(`*x*`)` returns `true` if *x* is odd; otherwise, it yields `false`.

The Abs function

`Abs(`*x*`)`

The `Abs` function returns the absolute value of a numeric value. Its parameter *x* is a numeric value. `Abs(`*x*`)` returns the absolute value of *x*, with the same type.

The Sqr function

`Sqr(`*x*`)`

The `Sqr` function returns the square of a numeric value. Its parameter *x* is a numeric value. `Sqr(`*x*`)` returns the square of *x*.

If *x* is of a `real` type , the result is `extended`; if *x* is of type `longint`, the result is `longint`; and if *x* is of type `integer`, the result may be either `integer` or `longint`.

If *x* is of `real` type and floating-point overflow occurs, the result is `+inf`. (See Appendix G and the *Apple Numerics Manual* for more information on infinities.)

The Sin function

`Sin(x)`

The `Sin` function returns an `extended` value that is the sine of a numeric value. Its parameter *x* is a numeric value. This value is assumed to represent an angle in radians. If *x* is infinite, a diagnostic NAN is produced and the invalid operation signal is set (see Appendix G).

The Cos function

`Cos(x)`

The `Cos` function returns an extended value that is the cosine of a numeric value. Its parameter *x* is a numeric value. This value is assumed to represent an angle in radians. If *x* is infinite, a diagnostic NAN is produced and the invalid operation signal is set (see Appendix G).

The Exp function

`Exp(x)`

The `Exp` function returns an extended value that is the natural exponential of a numeric value. Its parameter *x* is a numeric value. All possible values are valid. `Exp(x)` returns the value of e^x, where e is the base of the natural logarithm. If floating-point overflow occurs, the result is $+\infty$.

The Ln function

`Ln(x)`

The `Ln` function returns an extended value that is the natural logarithm of a numeric value. Its parameter *x* is a numeric value.

If *x* is nonnegative, `Ln(x)` returns the natural logarithm ($\log_e$) of *x*. If *x* is negative, a diagnostic NAN is produced and the invalid operation signal is set (see Appendix G).

The Sqrt function

`Sqrt (x)`

The `Sqrt` function returns an extended value that is the square root of a numeric value. Its parameter *x* is a numeric value.

If *x* is nonnegative, `Sqrt (x)` returns the positive square root of *x*. If *x* is negative, a diagnostic NAN is produced and the invalid operation signal is set (see Appendix G).

The Arctan function

`Arctan (x)`

The `Arctan` function returns an extended value that is the principal value, in radians, of the arctangent of a numeric value. Its parameter *x* is a numeric value. All numeric values of *x* are valid, including ±∞.

Ordinal functions

The ordinal functions operate on the ordinal value of scalar and pointer types, as explained in Chapter 4.

The Ord function

`Ord (x)`

The `Ord` function returns the ordinal number of a scalar type or pointer type value. Its parameter *x* is an expression with a value of scalar type or pointer type.

If *x* is of type `integer` or `longint`, the result is the same as *x*.

If *x* is of pointer type, the result is the corresponding physical address of type `longint`.

If *x* is of another scalar type, the result is the ordinal number determined by mapping the values of the type onto consecutive nonnegative whole numbers starting at zero.

For a parameter of type `char`, the result is the corresponding ASCII code. For a parameter of type `boolean`,

```
Ord(false) returns zero
Ord(true) returns one
```

The Chr function

```
Chr(x)
```

The `Chr` function returns the `char` value corresponding to a whole-number value. Its parameter *x* is an expression with an `integer` or `longint` value. `Chr(x)` returns the `char` value whose ordinal number (that is, its ASCII code) is *x*, if *x* is in the range 0..255. If *x* is not in the range 0..255, the value returned is not within the range of the type `char`, and any attempt to assign it to a variable of type `char` will cause an error.

For any `char` value `ch`, the following is true:

```
Chr(ord(ch)) = ch
```

The Succ function

```
Succ(x)
```

The `Succ` function returns the successor of its parameter *x*, a value of a scalar type. `Succ(x)` returns the successor of *x* if such a value exists according to the inherent ordering of values in the type of *x*.

If *x* is the last value in the type of *x*, it has no successor. In this case, the value returned is not within the range of the type of *x*, and any attempt to assign it to a variable of this type will cause unspecified results.

The Pred function

```
Pred(x)
```

The `Pred` function returns the predecessor of its parameter *x*, a value of a scalar type. `Pred(x)` returns the predecessor of *x* if such a value exists according to the inherent ordering of values in the type of *x*.

If *x* is the first value in the type of *x*, it has no predecessor. In this case, the value returned is not within the range of the type of *x*, and any attempt to assign it to a variable of this type will cause unspecified results.

String procedures and functions

The string procedures and functions do not accept `PACKED ARRAY OF char` parameters, and they do not accept indexed string parameters.

The Length function

`Length (`*str*`)`

The `Length` function returns an `integer` value that is the current length of its parameter *str*, which must have a value of type `STRING`.

The Pos function

`Pos (`*substr*, *str*`)`

The `Pos` function searches for *substr* within *str* and returns an `integer` value that is the index of the first character of *substr* within *str*. Both parameters must be of type `STRING`.

If *substr* is not found, `Pos` returns zero.

The Concat function

`Concat (`str_1, str_2, `[`str_3, `...,` str_n`])`

`Concat` concatenates all the parameters in the order in which they are written and returns the concatenated string. Character constants and strings may be mixed. Each parameter is an expression with a value of type `STRING`. Any number of parameters may be passed. Note that the number of characters in the result cannot exceed 255.

The Copy function

`Copy (`*source, index, count*`)`

`Copy` returns a string containing *count* characters from *source*, beginning at *source* [*index*]. The parameter *source* is an expression with a value of type `STRING`. The parameter *index* is an expression with an `integer` value in the range 1..255. The parameter *count* is an expression with an `integer` value in the range 1..255.

If the values of *index* or *count* are out of range or if there are not *count* characters in *source* starting at *source* [*index*], `Copy` returns the null string.

The Delete procedure

`Delete (`*dest, index, count*`)`

`Delete` removes *count* characters from the value of *dest*, beginning at *dest* [*index*]. The parameter *dest* is a variable reference that refers to a variable of type `STRING`. It is a variable parameter. The parameter *index* is an expression with an `integer` value in the range 1..255. The parameter *count* is an expression with an `integer` value in the range 1..255.

If the values of *index* or *count* are out of range or if *index* is greater than `Length (`*dest*`)`, `Delete` is ignored. If the attempted deletion extends beyond the end of *dest*, *dest* becomes truncated at *index* –1.

The Insert procedure

`Insert (`*source, dest, index*`)`

`Insert` inserts *source* into *dest*. The first character of *source* becomes *dest* [*index*]. The parameter *source* is an expression with a value of type `STRING`. The parameter *dest* is a variable reference that refers to a variable of type `STRING`. It is a variable parameter. The parameter *index* is an expression with an `integer` value in the range 1..255.

If the value of *index* is out of range, `Insert` is ignored.

Byte-oriented procedures and functions

The byte-oriented procedures allow a program to treat a program variable as a sequence of bytes, without regard to data types.

These procedures do no type checking on their *source* or *dest* actual parameters. However, because these are variable parameters, they cannot be indexed if they are packed. If an unpacked "byte array" is desired, then a variable of the type

```
ARRAY [lo..hi] OF -128..127
```

should be used for *source* or *dest*. The elements in an array of this type are stored in contiguous bytes; because it is unpacked, an array of this type can be used with an index as an actual parameter for these routines.

◆ *Note:* An unpacked array with elements of the type 0..255 or the type `char` has its elements stored in words, not bytes. A word is two bytes.

The Moveleft procedure

`Moveleft` (*source, dest, count*)

The `Moveleft` procedure copies a specified number of contiguous bytes from a source range to a destination range (starting at the lowest address). Its parameters are the following:

- *Source* is a variable reference that refers to a variable of any type except a file type or a structured type that contains a file type. It is a variable parameter. The first byte allocated to *source* (lowest address within *source*) is the first byte of the source range.
- *Dest* is a variable reference that refers to a variable of any type except a file type or a structured type that contains a file type. It is a variable parameter. The first byte allocated to *dest* (lowest address within *dest*) is the first byte of the destination range.
- *Count* is an expression with an `integer` value. The source range and the destination range are each *count* bytes long. The *count* parameter is not range checked.

The Moveright procedure

`Moveright` is exactly like `Moveleft`, except that it starts from the "right" end of the source range (highest address). It proceeds to the "left" (lower addresses), copying bytes into the destination range, starting at the highest address of the destination range.

The reason for having both `Moveleft` and `Moveright` is that the source and destination ranges may overlap. If they overlap, the order in which bytes are moved is critical: each byte must be moved before it gets overwritten by another byte.

The Sizeof function

`Sizeof(`*id* `[,` t_1`, ..., `t_n`])`

The `Sizeof` function returns a `longint` value that is the number of bytes occupied by a specified variable, or by any variable of a specified type. Its parameter *id* is either a variable identifier or a type identifier. The optional parameters $t_1, \ldots, t_n$ are tag values specified only to get the size of a variant record and may be specified only if the first parameter is a type identifier.

`Sizeof` returns the number of bytes occupied by *id*, if *id* is a variable identifier; if *id* is a type identifier, `Sizeof` returns the number of bytes occupied by any variable of type *id*. If the type *id* is a record that contains variants, you may specify the tag values (which must be constants listed contiguously and in order of their declaration). In this case, the `Sizeof` function returns the size of the record with the specified variants. The value of `Sizeof` is determined by the Compiler, which subsequently treats it as a constant at compile time.

Packed character array routines

The routines described in this section operate only on arrays of type `PACKED ARRAY OF char`. When used as parameters, such arrays cannot be subscripted; the routines described below always begin at their first character.

The Scaneq function

`Scaneq(`*limit, ch, paoc*`)`

`Scaneq` scans *paoc*, looking for the first occurrence of *ch*. The scan begins with the first character in *paoc*. If the character is not found within *limit* characters from the beginning of *paoc*, the value returned is an `integer` equal to *limit*. Otherwise, the value returned is an `integer` that gives the number of characters scanned before *ch* was found. The parameters of `Scaneq` have these types:

- *Limit* is an expression with a value of type `integer` or `longint`. It is truncated to 16 bits and is not range checked.
- *Ch* is an expression with a value of type `char`.
- *Paoc* is an expression with a value of type `PACKED ARRAY OF char`. It is a variable parameter.

The Scanne function

The `Scanne` function is exactly like `Scaneq`, except that it searches for a character that does not match the *ch* parameter.

The Fillchar procedure

`Fillchar(`*paoc, count, ch*`)`

The `Fillchar` procedure fills a specified number of characters in a `PACKED ARRAY OF char` with a specified character. It has the following parameters:

- *Paoc* is an expression with a value of type `PACKED ARRAY OF char`. It is a variable parameter.
- *Count* is an expression with a value of type `integer` or `longint`. It is truncated to 16 bits and is not range checked.
- *Ch* is an expression with a value of type `char`.

`Fillchar` writes the value of *ch* into *count* contiguous bytes of memory, starting at the first byte of *paoc*. Because the *count* parameter is not range checked, it is possible to write into memory outside of *paoc*, with unspecified results.

Logical bit functions and procedures

This section describes a set of procedures and functions for bit manipulations. These routines correspond to a set of essentially identical instructions of the Motorola 68000.

Many of the routines here correspond to *Inside Macintosh* routines. However, MPW Pascal generates more efficient code than calls to these routines, so you should use the identifiers given here in preference to the Macintosh ROM routines.

If the type of any argument is specified as a scalar, the argument can be a whole-number value of any size, from 1 to 32 bits (one bit to a long integer). If the scalar argument is less than 32 bits, code is generated to extend the argument to 32 bits but without sign extension (zeros are added on the left to make up a 32-bit value).

◆ *Note:* Bit numbering for these routines follows the convention of the 68000 microprocessor, not the convention used in *Inside Macintosh*. Bit 0 is the low-order bit; bit 31 is the high-order bit.

Table 11-1 summarizes the bit manipulation functions and procedures.

■ **Table 11-1** Bit manipulation routines

MPW name	MC68000 opcode	First argument	Second argument	Result	Kind
BAND	AND.L	scalar	scalar	longint	function
BOR	OR.L	scalar	scalar	longint	function
BXOR	EOR	scalar	scalar	longint	function
BNOT	NOT.L	scalar		longint	function
BSL	LSL.L	scalar	integer	longint	function
BSR	LSR.L	scalar	integer	longint	function
BRotL	ROL.L	scalar	integer	longint	function
BRotR	ROR.L	scalar	integer	longint	function
BTst	BTST.L	scalar	integer	boolean	function
HIWrd		scalar		integer	function
LOWrd		scalar		integer	function
BClr	BCLR.L	longint (VAR)	integer		procedure
BSet	BSET.L	longint (VAR)	integer		procedure

These routines are generally identical in function to a corresponding set of routines described in *Inside Macintosh*. However, these routines are more efficient because they are implemented by the Compiler as 68000 instructions, while the *Inside Macintosh* routines are calls to the ROM. The routines `BTst`, `BClr`, and `BSet`, while functionally similar to three *Inside Macintosh* routines, have different arguments.

The syntax of each bit manipulation routine is described below.

The BAND function

BAND (*arg1, arg2*)

BAND returns the logical AND of its two arguments.

The BOR function

BOR (*arg1, arg2*)

BOR returns the logical OR of its two arguments.

The BXOR function

BXOR (*arg1, arg2*)

BXOR returns the logical exclusive-OR of its two arguments.

The BNOT function

BNOT (*arg*)

BNOT returns the 1's-complement of its argument.

The BSL function

BSL (*arg, count*)

BSL left-shifts the bits of *arg* by the number of bits specified in *count*, modulo 64. Zeros are shifted into the low-order bit.

The BSR function

`BSR(`*arg*`, `*count*`)`

`BSR` right-shifts the bits of *arg* by the number of bits specified in *count*, modulo 64. Zeros are shifted into the high-order bit.

The BRotL function

`BRotL(`*arg*`, `*count*`)`

`BRotL` left-rotates the bits of *arg* by the number of bits specified in *count*, modulo 64. Bits shifted out of the high-order position go back into the low-order position.

The BRotR function

`BRotR(`*arg*`, `*count*`)`

`BRotR` right-rotates the bits of *arg* by the number of bits specified in *count*, modulo 64. Bits shifted out of the low-order position go back into the high-order position.

The BTst function

`BTst(`*arg*`, `*bitNbr*`)`

The parameter *bitNbr* is an `integer` that indicates the bit of *arg* to be tested. `BTst` returns `true` if the specified bit has the value one and returns `false` if it has the value zero. Because this function maps directly onto the 68000 instruction, bits are numbered in the way conventional in the 68000: 0 to 31, low-order bit to high-order bit.

The HiWrd function

`HiWrd(`*arg*`)`

`HiWrd` returns the high-order word of *arg*. If *arg* is not a `longint`, `HiWrd` returns zero. When the argument is a simple variable or array access, no code is generated by this function because the argument is simply addressed and used as an `integer`.

The LoWrd function

`LoWrd(`*arg*`)`

`LoWrd` returns the low-order word of *arg*. When the argument is a simple variable or array access, no code is generated by this function because the argument is simply addressed and used as an `integer`.

The BClr procedure

`BClr (`*arg*`, `*bitNbr*`)`

`BClr` clears bit *bitNbr* in *arg*. The value of *bitNbr* is reduced modulo 32.

The BSet procedure

`BSet (`*arg*`, `*bitNbr*`)`

`BSet` sets bit *bitNbr* in *arg*. The value of *bitNbr* is reduced modulo 32.

Chapter 12 **Object-Oriented Programming**

OBJECT-ORIENTED PROGRAMMING IS AN IMPORTANT FEATURE of MPW Pascal. This chapter briefly covers the philosophy behind object-oriented programming and the mechanisms built into MPW Pascal that support it. For a more complete treatment of object-oriented programming theory, see Kurt Schmucker's book *Object-Oriented Programming for the Macintosh,* listed in the Preface.

To use the object-oriented facilities of MPW Pascal, follow the instructions given under "Using Object Pascal" at the end of this chapter. ■

Contents

What are objects?

The object type is an addition to the familiar standard Pascal structured types. **Objects** are closely related to records; like a record, an object consists of a number of fields, each of which may be of a different type. Objects add an extra "dimension" to the idea of a record—they include not only data fields but also private procedures and functions (called **methods**). A method that is declared in an object type definition operates primarily on the data stored in an object of that type.

Much of the power of object-oriented programming derives from the concept of **inheritance.** You can define an object type as a customization of another object type. When one object type is derived from another, the first is called an **ancestor** and the second is called a **descendant.** A descendant type inherits all the fields and methods of its ancestor; you can add new fields and methods to it as well. Although you cannot change the interface to an inherited method, you can change the way it is implemented.

Here is an example of three object type declarations:

```
Employee = OBJECT
      firstName, lastName: STRING[32];
      hourlyWage, hoursPaid: integer;
      PROCEDURE Hire(name1, name2: string[32];
                     rate, hoursWorked: integer);
      FUNCTION RegularPay(hoursWorked: integer): integer;
      PROCEDURE IssuePaycheck(hoursWorked: integer)
END;

ExemptEmployee = OBJECT(Employee)
      FUNCTION RegularPay(hoursWorked: integer): integer;
      OVERRIDE;
END;

Executive = OBJECT(ExemptEmployee)
      weeklyBonus: integer;
      PROCEDURE SetBonus(performanceLevel: integer);
      PROCEDURE IssuePaycheck(hoursWorked: integer); OVERRIDE;
END;
```

In this example, the object type `Employee` begins the chain of inheritance. `Employee` isn't defined in terms of any other object—it doesn't have any ancestors. Like a record, an object of type `Employee` consists of several data fields. But in addition to data, an `Employee` object has three methods: the `Hire` and `IssuePaycheck` procedures, and the `RegularPay` function. These methods are declared as if they were `FORWARD` routines; their blocks come later in the program.

The object type `ExemptEmployee` is a descendant of `Employee`. In it, the function `RegularPay` is changed by overriding the function's body. However, it inherits unchanged all the data fields of `Employee`, as well as the procedures `Hire` and `IssuePaycheck`.

The object `Executive` is a descendant of `ExemptEmployee`. It retains data fields of `ExemptEmployee` (which are also fields of `Employee`) and adds the field `weeklyBonus`. It also adds a new procedure, `SetBonus`. In addition, it changes the `IssuePaycheck` method by overriding it. This lets `IssuePaycheck` operate on the value of `weeklyBonus`.

It is an important feature of this kind of programming that object type declarations do not need to include declarations that occur inside their ancestor objects; these declarations are automatically present by inheritance. They need to declare only changes. For example, the object types `ExemptEmployee` and `Executive` automatically include the procedure `Hire`, because they inherited it from `Employee`. As a result, you can create a very complex object type with a few simple declarations, just by naming another object type from which it inherits its structure.

Differences from traditional programming

You can look at the differences between object-oriented programming and standard programming in several ways:

- from a code viewpoint, in terms of the program structures that you create
- from a data viewpoint, in terms of the data structures they handle
- from a structural viewpoint, in terms of the resulting programming discipline

From a code viewpoint, each object in an object-oriented program may be thought of as a small virtual computer, existing independently within the overall computer. It operates on the data passing through it according to its own rules. To change these rules, you "reprogram" the object by changing its methods.

From a data viewpoint, the objects in an object-oriented program may also be thought of as "smart data structures." Each one not only stores information but also processes it, somewhat in the manner of a spreadsheet. Because each object operates on the information within it, you can treat its data fields as interrelated, rather than isolated.

Structurally, object orientation introduces a new kind of programming discipline. When creating a standard program, you "design down and code up." You first determine what blocks your program needs and then build the required blocks out of individual declarations and statements. Object-oriented programming lets you design down and code down at the same time. If you already have an object that nearly fits your requirements, you start coding with it. Instead of creating new blocks out of elemental parts, you create them by modifying existing blocks. You carve and shape, instead of piecing together. The object-oriented process is closer to high-level application programming and farther from machine programming. As a result, object-oriented programming has a number of practical advantages:

- You can use objects from existing programs to form new programs, instead of always building them anew. This is a great timesaver, particularly when you are developing large applications.
- When modifying objects, you start with something that already works and change its operation by easily understandable increments. This can decrease debugging time dramatically.
- Each object remains a closed universe; you don't need to worry about data leaks or code interactions between objects. This makes program development more orderly.
- In a complex environment, such as the Macintosh's, object-oriented programs are easier to maintain. Changes to the program yield relatively specific and predictable consequences.

Creating objects

You do not create objects from object types in the same way you create ordinary variables from other variable types. Instead, you use the standard `New` procedure to create an object of a given type. The `New` procedure sets aside a part of dynamic memory for the object, and returns a handle for the object.

- *Note:* This is an extension of the standard Pascal `New` procedure. When `New` is given an ordinary pointer variable, it reserves space for an identified variable and returns a pointer to the identified variable. When `New` is given an object type variable, it creates the object in dynamic memory and returns a handle to the object. See the Memory Manager chapter of ***Inside Macintosh*** for a discussion of handles.

Declaring object types

Unlike other types, an object type can only be declared in the type declaration part of a main program or unit. You cannot declare an object type in a variable declaration part or in a procedure or function declaration block.

When you use an object type in a variable declaration part, you create a reference variable for that type. The reference variable stores a handle to an object of that type (or a descendant type). You always access objects through reference variables. Hence you do not create special types for object references. Instead, an object type is used to declare each variable that can hold a reference to an object of that type (or a descendant type).

The scope of an object type (the type identifier and field and method identifiers) also extends over all descendants of that type, and over procedure and function blocks that implement methods of that object type and its descendants.

For an illustration of object type declarations, see the example given at the beginning of this chapter.

Object type membership

When an object of a specific type is created during program execution, it is considered a **member** of that type and of all ancestral types. In the example at the beginning of the chapter, an object created as an `Executive` is a member of types `Executive`, `ExemptEmployee`, and `Employee`. References to the object of type `Executive` may be assigned to object reference variables of types `Executive`, `ExemptEmployee`, and `Employee`. Which particular version of an overridden method is executed when a method call is executed depends on the type of the object, not the type of the reference variable.

Object reference variables

A variable that is declared using an object type is an **object reference variable.** An object reference variable is not itself an object. The value of an object reference variable is either NIL or a value that identifies an object, called the *identified object* of the reference. Objects themselves are dynamic variables. An object reference variable that refers to an object does so by means of a **handle**—a pointer to a pointer.

The pointer symbol (^) used to denote the identified variable of a pointer is not allowed after an object reference variable. Similarly, the double pointer symbol (^^) used to dereference a handle is not allowed. Hence there is no way to treat the identified object as a variable in its own right unless type coercion is used. However, you can access components of the object through a reference variable as you would access fields of a record.

Data is stored in fields of objects, and you access those fields by giving the reference variable identifier, a period, and the field name, which appears the same as a record field access. As with records, you can omit the reference variable identifier and period under certain circumstances. Variable accesses using the reference variable identifier alone access the pointer type value stored in the reference variable, just as with other pointers.

- *Note:* The MPW Compiler is now stricter about reporting errors about passing reference pointers to fields within objects.

The object reference variable and the period (.) can be omitted inside a WITH statement that lists the reference variable, or within any method block that declares a method of the object's type.

- *Note:* Using a WITH statement with an object reference variable does not dereference the handle that represents the object; however, the following three actions do dereference the object's handle: passing fields of an object as VAR parameters, passing fields longer than four bytes as parameters, or using a WITH statement for a field of an object that is itself a record (but not an object reference).

  Compaction of the heap can cause the object's handle to move and yield unpredictable results. The Pascal Compiler flags such unsafe object dereferences as errors unless the $H Compiler directive is turned off. (For details on the $H directive, see Chapter 13.)

Here is an example of how a variable declared as object type `Employee` can be used to refer to an object of type `Executive`, using the type declarations at the beginning of this chapter:

```
VAR anEmployee: Employee;
   anExecutive: Executive;
   {other declarations}
   New(anExecutive);
   anEmployee := anExecutive;
   anEmployee.IssuePaycheck(40);
```

The OVERRIDE directive

A descendant of an object type always inherits all fields and methods of its ancestors. It can add fields and methods to those it has inherited, and it can override the action of methods. To override a method, you follow the method heading with the word OVERRIDE. When a method is overridden, the implementation of the method is changed but the interface to the method must remain exactly the same. It must retain the same spelling of all identifiers and the same data types for the method's formal parameters and return value (if any).

Declaring methods

A method is a procedure or function that is declared as part of an object type declaration. Methods are declared like other procedures or functions, except that they are always declared in the style of a forward declaration but without the word FORWARD. Object types are often declared in the interface part of a unit, while the blocks of their methods are declared in the implementation part of the same unit.

An object type can inherit methods from another object type. If you want to override the action of an inherited method, write the word OVERRIDE following the formal parameter list (or following the method identifier, if there are no formal parameters). The formal parameter list you give for the override method must be identical to the formal parameter list of the overridden method. If you do not override an inherited method, you do not need to declare the inherited method.

When declaring a method, the initial heading declaration and the block declaration must both appear in the main program or must both appear in the same compilation unit. The heading declaration appears in the type declaration part, as part of the object type declaration, while the declaration of the method's block appears in the procedure and function declaration part. You must write the object type and a period along with the procedure or function identifier when you declare the block. You may repeat the formal parameter list; if you do, it must be identical to the original list.

The Self parameter

In addition to ordinary parameters, every method has an **implicit parameter,** called `Self`. `Self` is a reference to the object used to call the method. Its type is the reference type of the object type to which the method belongs. (Notice that this is not necessarily the type of the actual parameter supplied by the caller. `Self` could refer to an object of a descendant type.) The scope of `Self` extends over the method declaration block. You can assign values to fields of `Self`, but you cannot change the value of `Self`, which would cause `Self` to reference another object. The method acts as if its entire statement part were surrounded by `WITH Self DO BEGIN ... END`, so you do not have to give the identifier `Self` when accessing fields or methods of `Self`.

This object type definition uses the implicit parameter `Self` in a method declaration:

```
TYPE AnObject = OBJECT
 PROCEDURE Grow(howBig: integer)
 END;
 {other type declarations}
 PROCEDURE ThisObjectGrew(obj: AnObject; howMuch: integer);
 BEGIN
 ...
 END;
 PROCEDURE AnObject.GRow(howBig: integer);
 BEGIN
 ThisObjectGrew(SELF, howBig);
 END;
```

Calling methods

A method call is a special case of a function call. The same rules apply to methods and method calls for both procedures and functions.

As with fields of objects or records, a method can be accessed by using a qualifier or by using a `WITH` statement.

Here are some examples of procedural method calls:

```
v.Draw
WITH v DO Draw
x[i].Track(y+1, z-1)
```

Here are some examples of functional method calls:

```
a.Times(b) + c
y: = x.Extent
WITH x DO y := Extent
```

Unlike a field, which can be evaluated or assigned a new value, a method is executed when it is called. When a method is called, a reference to the specific object through which the method was accessed (for example, `object v` in `v.Draw`) is bound to the automatically declared formal parameter `Self`, whose type is the object type of which the method is part.

If `y` is declared to be a variable of object type `TView`, then at any moment during execution `y` may be a reference to an object of any type that inherits from `TView`. If `TView` has a method `Draw`, for example, different inheriting types may define different implementations of the method `Draw`. Executing `y.Draw` calls the implementation of `Draw` defined for the type of the object that `y` refers to at the time of the call. The value of `Self` in the called method becomes a reference to that same object. Note that `y` may be a variable of a different type—the method actually called is the one that belongs to the object, not the variable.

One way to read the statement `y.Draw` is "Tell `y` to draw." Reading it that way points out that the program only says the object referred to by `y` should draw. At run time, the current value of `y` determines how it should draw by choosing the appropriate implementation of draw for its object type.

The INHERITED directive

When one object type inherits from another but overrides an inherited method with its own implementation, you may want to call the overridden method from within the new method.

For example, if object type `Truck` inherits from `Vehicle` and overrides method `Accelerate`, then `Truck.Accelerate` may wish to invoke `Vehicle.Accelerate` at one or more points. You can call the overridden method with the special form

```
INHERITED Accelerate
```

In general, to call a method that belongs to the immediate ancestor of the object type that owns the current method, write `INHERITED` (without a period). The value of `Self` in the called method is the same value as it is in the calling method.

`INHERITED` may only be used within a method declaration block. It must precede a method identifier that was inherited by the object type that owns the method declaration block.

Using Object Pascal

There are two basic ways you can write object-oriented applications for the Macintosh: without or with MacApp. This section discusses both approaches.

Object Pascal without MacApp

Support for Object Pascal syntax is included in the MPW Compiler, so you do not need MacApp to write an object-oriented program. However, you do need to provide runtime support by linking your compilation with the file ObjLib.o.

You must also access the unit ObjIntf.p by including the declaration `USES ObjIntf.p` in your source text. ObjIntf.p provides the interface for the type `TObject`, which has no data fields and four methods—`ShallowClone`, `Clone`, `ShallowFree`, and `Free`. These routines are described below.

The Object Pascal routines

This section describes five functions available for use in object-oriented programming without MacApp.

The Member function

`Member(`*anObject*, *aType*`)`

`Member` is used only in object-oriented programming. `Member` tests if a particular object is of a particular object type or a descendant of that type. It has two parameters:

- *AnObject* is an object reference. It is an error if *anObject* is undefined. *AnObject* can have the value `NIL`.
- *AType* is an object type.

`Member` returns `true` if *anObject* is not `NIL` and if the object it references is a member of the type *aType*. The parameter *anObject* is a member of *aType* if it is of that type or a descendant of *aType*.

Although rarely used, `Member` is useful for screening questionable object reference coercions. The following use of member is strongly discouraged, however, because it defeats the advantages of object-oriented programming:

```
IF member(x, A) THEN…
ELSE IF member(x, b) THEN…
ELSE IF member(x, c) THEN…
```

Instead, define the code in each `THEN` clause as a method of the corresponding types. Give all the methods the same name and arguments. Be sure their common ancestor also declares the method. Write a method call instead of the entire conditional statement. That way, new types can be added without changing the program, which is one of the main advantages of object-oriented programming.

The ShallowClone function

`ShallowClone`

`ShallowClone` returns a copy of an object of type `TObject`. It is a function without any parameters. You should not override it. If you want to override it to copy objects referred to by fields, use `Clone`, described below.

The Clone function

`Clone`

`Clone` normally calls `ShallowClone`, but may be overridden to copy objects referred to by fields.

The ShallowFree function

`ShallowFree`

`ShallowFree` frees the space occupied by an object in memory. You should not override it. If you want to override it to free objects referred to by fields, use `Free`, described below.

The Free function

`Free`

`Free` normally calls `ShallowFree`, but may be overridden to free objects referred to by fields.

Object Pascal with MacApp

The simplest way to create an object-oriented application is to use MacApp, Apple's "expandable" Macintosh application. MacApp implements the Macintosh user interface in an object-oriented environment. To create a specific application to fit your requirements, you expand MacApp by adding to it a series of descendant object type declarations, each of which inherits some part of the original object type. These descendant objects become the elements of your program.

From the outset, each new object type inherited from MacApp is guaranteed to work because its ancestor object type has already been debugged. Its data and methods form an integrated whole—a virtual computer within your program that performs a specific group of tasks. As you modify it to meet your needs, you can test each modification and see its effects. This helps you achieve orderly and bug-free program development. MacApp frees you from many of the chores required when you write programs from the ground up. It lets you concentrate on the parts of your application that are specific to the job it performs.

MacApp provides built-in runtime support for object-oriented programming; you do not need to link your program to any other file.

Chapter 13 **Compiler Options and Directives**

THE COMPILER CAN BE CONTROLLED IN THREE WAYS: by using the Compiler options available from the MPW command line, by using Compiler directives that you write directly in your Pascal source text, or by using the Commando tool from the MPW Shell. The Commando tool is a series of dialog boxes that displays all the functions, parameters, and options for MPW commands, including Pascal. See the *Macintosh Programmer's Workshop 3.0 Reference* for a discussion of the interface available for Pascal and for information on writing your own series of dialog boxes. ■

Contents

The MPW Pascal command line

This is the syntax for specifying options on the Pascal command line:

`Pascal`[*option...*][*file...*]

You can specify zero or more filenames. Each file is compiled separately—compiling file *Name*.p creates object file *Name*.p.o. By convention, Pascal source filenames end in a ".p" suffix. If you do not specify a filename, standard input is compiled to a file named "p.o".

Compiler errors are written to diagnostic output, a predeclared file of type `text`, which can be written to another file or redirected. Progress and summary information is also written to diagnostic output, if requested, by using the Compiler directives described in the section of that name. Diagnostic output is fully described in the *Macintosh Programmer's Workshop 3.0 Reference.*

Compiler options

The MPW Pascal Compiler options are symbols in the MPW command line that send instructions to the Compiler. MPW 3.0 Pascal supplies an option, `-sym`, that emits records for the symbolic debugger and an option, `-mbg`, that includes symbols for MacsBug. Also, the `-h` option suppresses error messages regarding the use of unsafe handles, the `-m` option allows greater than 32k globals, the `-k` option puts symbol table resources in the directory specified by `prefixpath` and the `-n` option generates separate global data modules for better allocation. Finally, three new command line options, `-noload`, `-clean`, and `-rebuild`, support the Compiler's automatic loading facility. MPW 3.0 Pascal no longer offers the `-z` option. Table 13-1 presents an alphabetical listing of the Compiler options for MPW 3.0 Pascal.

■ **Table 13-1** Compiler options

Option	Description
`-b`	Generate A5-relative references whenever the address of a procedure or function is required. (By default, PC-relative references are generated for routines in the same segment.) This option is equivalent to specifying `{$B-}` in the source code.
`-c`	Syntax check only—no object file is generated.
`-clean`	Erase all symbol table resources.
`-d` *name*`=true | false`	Set the compile-time variable *name* to `true` or `false` (for example, `-d Elems881=true` directs the Compiler to emit direct calls to the 68881 for transcendental functions. This option is equivalent to specifying `{$SETC name:=true|false}` in the source code.
`-e` *file*	Write all errors to the error log file *file*. A copy of the error report will still be sent to diagnostic output. This option is equivalent to specifying `{$E` *file*`}` in the source code.
`-h`	Suppress error messages regarding the use of unsafe handles.
`-i`*pathname*`[,` *pathname*`]...`	Search for include or `USES` files in the specified directories. Multiple `-i` options may be specified. At most, 15 directories will be searched. The search order is as follows: 1. In the case of a `USES` filename, if no prior filename was specified, the filename is assumed to be the same as the unit name (with a ".p" appended). 2. The filename is used as specified. If a *full pathname* is given, no other searching is applied. If the file is not yet found and the pathname used to specify the file is a *partial pathname* (no colons in the name or a leading colon), the following directories are searched: 3. The directory containing the current input file. 4. The directories specified in `-i` options, in the order listed. 5. The directories specified in the Shell variable `{PInterfaces}`.

(Continued)

■ **Table 13-1** (Continued) Compiler options

Option	Description
`-k`	Put symbol table resources in the directory specified by `prefixpath`.
`-m`	Allow greater than 32K globals by using 32 bit references.
`-mbg ch8`	Include v2.0 compatible MacsBug symbols (eight characters only, in a special format).
`-mbg full*`	Include full (untruncated) symbols for MacsBug.
`-mbg off`	Don't include symbols for the MacsBug debugger.
`-mbg` *number*	Include MacsBug symbols truncated to length *number*.
`-MC68020`	Generate 68020 code. The Compiler generates 68020 instructions or addressing modes that are selected to be faster and/or smaller than the 68000 equivalent. When the Compiler is generating 68020 code for the main program, it inserts a `TRAPF` instruction in the program preamble code. The `TRAPF` instruction does nothing on the 68020 and causes an illegal instruction trap on a 68000 so that your programs can detect early whether they are being run without a 68020. This option is equivalent to specifying `{$MC68020+}` in the source code.
`-MC68881`	Generate 68881 code on a per-file basis. The Compiler allocates 12 bytes for each `extended` variable and assigns up to 4 `extended` variables (either local or parameter) per procedure to the 68881 registers FP4 through FP7. The Compiler generates 68881 code whenever possible for arithmetic operations and binary data (but not string) conversions. When the Compiler generates code to call an external routine, the code passes parameters using the MPW Pascal 1.0 conventions with one exception: `extended` types are 12 bytes wide. However, the Compiler expects that a C function returning an `extended` type will return the result value in register FP0.

(Continued)

■ **Table 13-1** (Continued) Compiler options

Option	Description
	If `-MC68881` and `-d Elems881=true` are both specified on the command line, then the Compiler also recognizes and generates inline code for the `Sin`, `Cos`, `Ln`, `Exp`, `Arctan` routines, and these additional routines that are supported by the 68881: `Arctanh`, `Cosh`, `Sinh`, `Tanh`, `Log10`, `Exp10`, `Arccos`, `Arcsin`, `Sincos`, `Tan`, `Exp1`, `Log2`, The `-MC68881` option is equivalent to specifying `{$MC68881+}` in the source code.
`-n`	Generate separate global data modules for better allocation.
`-noload`	Don't use or create any symbol table resources.
`-o` *objname*	Specify the pathname for the generated object file. If *objname* ends with a colon (`:`), it indicates a directory for the output file, whose name is then formed by the normal rule (that is, *inputFilename*.o). If the source filename contains a pathname, it is stripped off and replaced by *objname:* as a prefix. (In this case, only one source file should be specified.) If *objname* does not end with a colon, the object file is written to the file *objname*.
`-ov`	Turn on overflow checking. (*Warning:* This may significantly increase code size.) This option is equivalent to specifying `{$ov+}` in the sourcecode.
`-p`	Supply progress and summary information to diagnostic output, including Compiler header information (copyright notice and version number), module names and code sizes in bytes, and number of errors and compilation time.
`-r`	Suppress range checking. This option is equivalent to specifying `{$R-}` in the source code.
`-rebuild`	Rebuild all symbol table resources.
`-sym off*`	Don't emit SADE object file information.
`-sym on | full`	Emit complete SADE object file information. To limit this option, also specify one or more of `novars`, `nolines`, `notypes` to omit variable, line, or type information respectively from the object file.

(Continued)

■ **Table 13-1** (Continued) Compiler options

Option	Description
`-t`	Report compilation time to diagnostic output. The `-p` option also reports the compilation time.
`-u`	Initializes all data (global and local) to the pattern `$7267`. This option is useful for debugging programs that may be using uninitialized data.
`-w`	Halt the operation of the peephole optimizer on a per-file basis. The Compiler no longer executes a final pass on the generated code and no longer attempts to replace certain code sequences with more efficient ones. This option is equivalent to specifying `{$W-}` in the source code.
`-y` *pathname*	Put the Compiler's intermediate (".o.i") files in the directory specified by *pathname*.

Compiler directives

Compiler directives are commands that you embed directly in the Pascal source code.

Every Compiler directive begins with a dollar sign ($) and must be enclosed in comment delimiters, as described under "Comments and Compiler Directives" in Chapter 2. You can put only one directive within each pair of delimiters.

The MPW Pascal Compiler directives are listed alphabetically in Table 13-2, with the default conditions marked by asterisks. The individual directives are discussed in the remainder of this chapter.

The MPW 3.0 Pascal Compiler supports the `$K` directive which puts symbol table resources in the directory specified by `prefixpath`.

MPW 3.0 Pascal no longer offers the `$LOAD` Compiler directive. The `$LOAD` syntax is still supported, but ignored—if Compiler progress information is requested, the Compiler states that the use of the feature is "obsolete but harmless." If you have included dependencies for `$LOAD` files in your makefiles, you can remove them; however, if you do not remove them, they remain harmless because they simply restate what the Compiler does automatically.

■ **Table 13-2** Compiler directives

Directive	Effect
`$A1`	Allow the global data sections of the unit to be noncontiguous
`$A5`	Let Compiler resolve references to the unit's global data
`$B+`	Generate PC-relative code*
`$B-`	Generate A5-relative code
`$C+`	Generate code*
`$C-`	Do not generate code
`$D-`	Do not embed routine names in object code
`$D+`	Embed routine names in object code*
`$E` *filename*	Send compilation errors to *filename* (See detailed discussion of filenames and segment names below.)
`$ELSEC`	Compile source text if *comp-expr* in preceding `$IFC` is `false`
`$ENDC`	End range of conditionally compiled source text
`$H+`	Check dereferencing of object handles*
`$H-`	Assume all object handles are valid
`$I` *filename*	Include separate source file in the compilation (See detailed discussion of filenames and segment names below.)
`$IFC` *comp-expr*	Compile subsequent source text if value of *comp-expr* is `true`
`$IFC OPTION` (*option-name*)	This version of `$IFC` lets you determine the current settings of Compiler options
`$J-`	Global data definitions must be in the Pascal source file*
`$J+`	Global data may be defined in another file
`$K` [*pathname*]	Control destination of symbol table resources
`$MC68020+`	Generate 68020 code on a per-procedure basis
`$MC68020-`	Halt generation of 68020 code on a per-procedure basis*
`$MC68881+`	Generate 68881 code on a per-file basis
`$MC68881-`	Halt generation of 68881 code on a per-file basis*

(Continued)

■ **Table 13-2** (Continued) Compiler directives

Directive	Effect
`$N-`	Identify all routines to the Linker as anonymous*
`$N+`	Send actual names of routines to the Linker
`$OV-`	Ignore arithmetic overflows*
`$OV+`	Detect arithmetic overflows
`$P`	Tell PasRef to do a page eject
`$PUSH`	Save the current option settings
`$POP`	Restore the saved option settings
`$R+`	Perform range checking of strings, sets, and arrays*
`$R-`	Do not perform range checking of strings, sets, and arrays
`$SC-`	Normal evaluation of AND and OR*
`$SC+`	Short-circuit evaluation of AND and OR
`$S` *segname*	Place subsequent routines in segment *segname* (See detailed description of *filename/segname* below.)
`$SETC` *id* := *comp-expr*	Declare a compile-time variable and assign it a value
`$U` *filename*	Specify *filename* for next unit in USES declaration (See detailed description of *filename/segname* below.)
`$W+`	Turn on the peephole optimizer*
`$W-`	Halt the operation of the peephole optimizer
`$Z-`	Identify all routines and variables to the Linker as local*
`$Z*`	Identify all routines at the top level to theLinker as external
`$Z+`	Identify all routines and variables to the Linker as external

Input file control

The $I directive

{$I *filename*}

The $I directive instructs the Compiler to fetch subsequent source input from the specified file. The Compiler will read from the new file until the end of that file, at which point the Compiler will continue from the original source file. The filename can include prefixes if desired; however, the Compiler will open the file by using its search rules. Included files may be nested up to five deep.This number includes units accessed by USES declarations in included files or nested units. A unit may not use the $I directive in its interface section. See "Shell Variable Substitution in Filenames and Segment Names" below for more details on the $I directive.

The $U directive

{$U *filename*}

In a USES statement, the Compiler will read each unit name and will search for the corresponding file *unitname*.p. This mechanism can be overridden by using the $U command to specify a filename in which to find the following unit. In either case, the Compiler will use its search rules to open the file. Each $U directive is valid only once, for the next unit name specified in the source text. See the section "Shell Variable Substitution in Filenames and Segment Names" below for more details on the $U directive.

Shell variable substitution in filenames and segment names

Four directives ({$E}, {$I}, {$S}, and {$U} require a string that is interpreted as a filename (or in the case of {$S} as a segment name). For these directives, the value of the string may be controlled by an MPW Shell variable. In the following example, {$I} is used, but the discussion holds for all five directives.

Normally, the directive has the form {$I *string*} in which case the string is exactly as specified. For example, {$I foo.p} includes the file foo.p.

Two other possibilities are

- `{$I $$Shell(`*shell-variable*`)}`, in which case the value of the string is the value of the specified shell variable. For example, `{$I $$Shell(myVar)}` is equivalent to `{$I foo.p}` if the shell variable `myVar` is set to `foo.p`.
- `{$I $$Shell(`*shell-variable*`) string}`, in which case the value of the string is the value of the specified shell variable with the value of the specified string appended. For example, `{$I $$Shell(myVar)foo.p}` is equivalent to `{$I hd:includes:foo.p}` if the shell variable `myVar` is set to `'hd:includes:'`.

It is an error if the Compiler cannot access a specified shell variable.

One way to specify the shell variable `myVar` in the above example would be

```
set myVar 'hd:includes:'
export myVar
```

See the *Macintosh Programmer's Workshop 3.0 Reference* for further information about shell variables.

Control of code generation

The $B± directive

```
{$B+}
{$B-}
```

When a program takes the address of a routine (for example, with the `@` operator) that is in the same segment, the Compiler generates PC-relative code. The directive `$B-` forces the Compiler to generate A5-relative code instead. The default value `$B+` switches back to PC-relative code.

The $C± directive

```
{$C+}
{$C-}
```

When this command is turned off (`$C-`), the Compiler will not produce object code for subsequent statements, although syntax checking still continues. Code generation can be resumed by specifying `$C+`. `$C+` is the default condition. This directive affects only entire procedures or functions.

The $J± directive

```
{$J+}
{$J-}
```

The directive `$J+` allows global data declared in the source file to be defined in another file, the connections being made by the Linker. Such connections are case sensitive. The default directive `$J-` requires all definitions to be in the Pascal source file.

The $MC68020± directive

```
{$MC68020+}
{$MC68020-}
```

The directive `$MC68020+` permits the Compiler to generate 68020 code. The default directive `$MC68020-` halts the generation of 68020 code.

The $MC68881± directive

```
{$MC68881+}
{$MC68881-}
```

The directive `$MC68881+` permits the Compiler to generate 68881 code on a per-file basis. The default directive `$MC68881-` halts the generation of 68881 code on a per-file basis.

The $OV± directive

```
{$OV-}
{$OV+}
```

The default condition `$OV-` prevents the Compiler from generating code to detect arithmetic overflow during assignments and expression evaluation. The directive `$OV+` causes it to produce such code.

The $R± directive

```
{$R+}
{$R-}
```

The `$R-` command instructs the Compiler to forego the generation of code to perform range checking of string, set, and array bounds. The default, `$R+`, is to produce such code.

The $S directive

```
{$S [segname]}
```

By default, the Compiler will instruct the Linker to place all routines within a single segment (with the case-sensitive identifier `Main`). The `$S` command allows the programmer to specify that subsequent routines be directed to the specified segment. The `$S` command can only appear between global routines. If *segname* is omitted, the segment name `Main` is assumed.

The $SC± directive

```
{$SC-}
{$SC+}
```

The `$SC+` directive instructs the Compiler to evaluate AND and OR as short-circuit operators. In this case, the evaluation process starts with the left operand and ends when a `true` value has been reached for the expression. The default directive `$SC-` causes the Compiler to evaluate both operands of AND and OR.

The $W± directive

```
{$W-}
{$W+}
```

The default directive `$W+` turns on the peephole optimizer. The `$W-` directive halts the operation of the peephole optimizer.

Debugging

The $D± directive

```
{$D+}
{$D-}
```

The Macintosh debugger MacsBug is capable of reading routine names embedded in the object code. By default, the Compiler embeds the procedure (or function) name in the object code. `$D-` turns off this feature; `$D+` turns it back on.

The $H± directive

```
{$H+}
{$H-}
```

When the default value `$H+` is in effect, the Compiler tests each expression that dereferences a handle in object-oriented source text, to make sure it is currently valid. It issues a Compiler error if it is not. The directive `$H-` disables such verification.

Conditional compilation

MPW Pascal lets you compile sections of your source text conditionally by means of the `$IFC`, `$ELSEC`, and `$ENDC` directives. The `$IFC` directive is controlled by the value of a **compile-time expression.** Sections of source text controlled by these directives may be nested; the rules covering such nesting are the same as the rules for `IF` statements explained in Chapter 7.

You can form compile-time expressions out of **compile-time variables** or constants of type `integer` or `boolean`. The final value of the expression that controls a `$IFC` directive must be `boolean`. You can use all the Pascal operators in compile-time expressions except `IN` and `@`. If you use the operator `/`, the Compiler will automatically change it to `DIV`.

The $SETC directive

`{$SETC` *id* `:=` *comp-expr*`}`

The `$SETC` directive declares a compile-time variable named *id* and assigns it the value *comp-expr.*

The $IFC directive

`{$IFC` *comp-expr*`}`

The `$IFC` directive causes the Compiler to compile subsequent source text until the next `$ELSEC` or `$ENDC` directive, only if the `boolean` value of *comp-expr* is `true`.

You can write this directive in the form `{$IFC UNDEFINED` *varname*`}`. This will act like `{$IFC true}` if *varname* has not yet been declared with a `$SETC` directive; otherwise, it will act like `{$IFC false}`.

You can also write this directive `{$IFC OPTION(`*option-name*`)}` to test the current setting of Compiler options. For example, `{$IFC OPTION (MC68881)}` acts like `{$IFC true}` if `{$MC68881+}` had been specified prior to the $IFC.

The $ELSEC directive

```
{$ELSEC}
```

The `$ELSEC` directive marks the beginning of source text that is compiled only if the value of the *comp-expr* controlling its corresponding `$IFC` directive is `false`.

The $ENDC directive

```
{$ENDC}
```

The `$ENDC` directive marks the end of a section of conditionally compiled source text, matching a `$IFC` directive.

Output control

The $Z± directive

```
{$Z*}
{$Z+}
{$Z-}
```

The Compiler identifies all routines and variables (other than those in a unit interface) to the Linker as local (that is, not accessible from outside the program). By specifying `$Z+`, the programmer will force the Compiler to identify subsequent routines and variables as external. The `$Z-` command returns the Compiler to its default action. A subset of `$Z+` is available: `$Z*` forces the Compiler to identify subsequent routines at the top nesting level (and not variables) as external. Using `$Z+` increases the size of the code file substantially.

The $N± directive

```
{$N+}
{$N-}
```

By default, all routines are identified to the Linker as anonymous. When `$N+` is specified, the Compiler passes the actual names of subsequent routines to the Linker. This can be useful for tracking down link-time errors, as the Linker will be able to report the name of the routine involved in the error. `$N-` returns the Compiler to its default behavior.

- *Note:* The `$N±` directive operates on an entire source file; `$Z±` can be used in selected areas of a source file.

Other directives

The $A1 directive

{$A1}

By default, the global data for the interface and implementation sections of a unit is allocated contiguously in memory. By specifying $A1 (1 is one) in the interface of a unit, the programmer can allow the two data sections to be allocated noncontiguously.

The $A5 directive

{$A5}

Normally, the global data of the main program is located immediately below register A5, and all data references within this area are resolved by the Linker. By specifying $A5 in a unit (before any interface data declarations), the programmer ensures that the unit data is located immediately below register A5 and that references to that unit's data are resolved by the Compiler. This is useful when most global data is declared in a separate unit. The $A5 directive can only be used in a unit and only in one unit within a compilation.

The $E directive

{$E *filename*}

When the $E command is used, subsequent compile-time errors are sent to the file specified; they are also echoed to standard diagnostic. By default, compile-time errors are not sent to an error file. The filename specified must include the necessary prefixes; none will be supplied by the Compiler. See the section "Shell Variable Substitution in Filenames and Segment Names" above for more details on the $E directive.

The $K directive

{$K[*dirname*]}

Normally, the symbol table resources of a unit will be stored in the resource fork of the unit's source file. With the $K directive, if a directory is specified, symbol table resources will be stored in a file with the same name as the unit it came from, but in the directory *dirname*. If no directory is specified, symbol table resources will be stored in the unit's source file. The $K directive differs from the -k compiler option in that it allows you to specify which units will be stored in which directory. You may find the $K directive useful if your units are in a location that cannot be written to.

For example, you might use the $K directive while reading units from a file server. If you give a $K directive and a directory name, the Compiler will read the source code on the file server for the unit it is looking for. Then the Compiler creates a file with the same name as the unit in the specified directory and will store and read the symbol table resources there.

The $P directive

```
{$P}
```

The $P directive tells PasRef to perform a page eject. For further information about PasRef, see Appendix I.

The $PUSH and $POP directives

```
{$PUSH}
{$POP}
```

The $PUSH directive allows you to save the current option settings.

The $POP directive allows you to restore the saved option settings.

These directives are used with includes or with USE statements. It is an error to have more {$POP}s than {$PUSH}es.

Appendix A **MPW 3.0 Pascal and Other Pascals**

THIS APPENDIX CONTAINS BRIEF DESCRIPTIONS of the differences between MPW 3.0 Pascal, ANS Pascal, and MPW 2.0 Pascal. ■

Contents

MPW 3.0 Pascal and ANS Pascal

MPW 3.0 Pascal contains several exceptions and extensions to American National Standard (ANS) Pascal, as described below.

Exceptions to the ANSI Standard

The MPW 3.0 Pascal Compiler complies with the requirements of ANSI/IEEE770X3.97-1983, with the following exceptions:

- Identifiers are limited to 63 characters.
- The at symbol (`@`) is not equivalent to the caret (`^`).
- Values that are assigned to pointers can be obtained in ways other than from the `New` procedure.
- The range of a `SET OF integer` is limited to 0..2039.
- The MPW 3.0 Pascal string type is stored as a one-byte-length field followed by the characters in the string. The ANSI Standard string type is a `PACKED ARRAY[1..n] OF char`.
- In MPW 3.0 Pascal, the type `text` is distinct from the type `FILE OF char`. The type `FILE OF char` is a file whose records are of type `char`, containing `char` values that are not interpreted or converted in any way during I/O operations.
- The procedures `Pack` and `Unpack`, described by Jensen and Wirth, are not supported.
- The Standard comment delimiters {} and (**) are used to allow comment nesting, so the Standard comment delimiters {**} and (**) are not supported.

Extensions to ANS Pascal

In addition to the requirements of the Standard, this implementation of Pascal includes the following extensions:

- Constant expressions are allowed in declarations and indexes.
- Declarations can be written in any order.
- The `Cycle` statement is supported.
- The `Leave` statement is supported.
- Ranges are allowed in `CASE` statement tags.
- The Standard Apple Numeric Environment is supported.
- The `Exit` procedure is supported.
- The vertical bar (`|`) and the ampersand (`&`) operators are supported.
- Functions can return values of structured types.
- `Univ` parameters are supported.
- Type coercion techniques are supported.
- There are built-in bit manipulation routines.
- The exponentiation operator `**` is supported.
- Units and the `USES` declaration are supported.
- Object Pascal is supported.
- The predefined constants `maxlongint`, `pi`, `inf`, `maxcomp`, `minnormreal`, `minnormdouble`, `minnormextended`, `compsecs`, `compdate`, and `comptime` are supported.
- The routines `open`, `blockread`, `blockwrite`, `byteread`, and `bytewrite` are supported.
- The routines `Arctanh`, `Cosh`, `Sinh`, `Tanh`, `Log10`, `Exp10`, `ArcCos`, `Arcsin`, and `Sincos` are supported.

Implementation-dependent features

ANSI/IEEE770X3.97-1983 defines several requirements that are implementation-dependent. The Standard uses the term *implementation-dependent* to describe a feature that may differ between processors but that is not necessarily defined for any particular processor.

The effect of using a feature of MPW 3.0 Pascal that is required by the Standard, but that is implementation-dependent, is unspecified. Programs that use these features should not depend on any specific course being chosen because the results may be unpredictable. This leaves MPW free to choose the course that is most convenient at the time.

MPW 3.0 Pascal and MPW 2.0 Pascal

MPW 3.0 Pascal differs from MPW 2.0 Pascal in the following ways:

- The MPW 3.0 Pascal Compiler no longer provides the command line option `-z` or the Compiler directive `$LOAD`.
- The MPW 3.0 Pascal Compiler provides an automatic replacement for the `$LOAD` mechanism.
- The MPW 3.0 Pascal Compiler provides a directive (`$K`) that controls the destination of symbol table resources.
- The MPW 3.0 Pascal Compiler provides the command line options `-sym`, `-mbg`, `-noload`, `-clean`, `-rebuild`, `-k`, `-h`, `-m`, and `-n`.
- The MPW 3.0 Pascal Compiler provides support for greater than 32K global data.
- The MPW 3.0 Pascal Compiler imposes less strict requirements for forward class references.
- The MPW 3.0 Pascal Compiler allows character constants as valid string expressions.
- The MPW 3.0 Pascal Compiler extends the ability to include symbols for MacsBug.

Appendix B **Special Scope Rules**

THIS APPENDIX DESCRIBES CERTAIN SCOPE RULES of MPW Pascal that are applicable under special circumstances. ■

Contents

Scope of enumerated scalar constants

Consider the following program:

```
PROGRAM Cscope1;
 CONST ten = 10;
 PROCEDURE P;
  CONST ten = ten; {This should be an error.}
  BEGIN
   Writeln(ten)
  END;
 BEGIN
  P
 END.
```

The constant declaration in procedure `P` should cause a Compiler error, because it is illegal to use an identifier within its own declaration (except for pointer identifiers). However, the Compiler does not detect errors of this kind. It assigns the value of the global constant `ten` to the local procedure constant `ten`; the `Writeln` statement therefore writes the number 10.

A more serious anomaly of the same kind is illustrated by the following program:

```
PROGRAM Cscope2;
 CONST red = 1;
    violet = 2;
 PROCEDURE Q;
  TYPE arrayType = ARRAY[red..violet] OF integer;
     color = (violet, blue, green, yellow, orange, red); {Error?}
  VAR arrayVar: arrayType;
     c: color;
  BEGIN
   arrayVar[1] := 1;
   c := red;
   Writeln(Ord(c))
  END;
 BEGIN
  Q
 END.
```

Within the procedure `Q`, the global constants `red` and `violet` are used to define an array index type, making `ARRAY[red..violet]` equivalent to `ARRAY[1..2]`. In the declaration of the type `color`, the constants `red` and `violet` are locally redefined and given the new ordinal values of five and zero. Hence the `Writeln` statement writes the number 5.

Using `red` in the declaration of `color` should cause a Compiler error, but it does not.

If the first statement of the main program, `arrayVar[1] := 1`, is replaced by the statement `arrayVar[red] := 1`, a Compiler error will result because `red` is now an illegal index value for `arrayVar`—even though `arrayVar` is of type `arrayType`, which is defined as `ARRAY[red..violet]`.

To avoid this kind of problem, do not redefine constant identifiers of enumerated scalar types.

Scope of pointer base types

Consider the following program:

```
PROGRAM Pscope1;
 TYPE s = 0..7;
 PROCEDURE Makecurrent;
  TYPE sptr = ^s;
     s = RECORD
       ch: char;
       bool: boolean
      END;
  VAR current: s;
     ptrs: sptr;
  BEGIN
   New(ptrs);
   ptrs^ := current  {Compiler error here}
  END;
 BEGIN
  Makecurrent
 END.
```

This program declares a global integer subrange type `s` and also a local record type `s`. Within the procedure `Makecurrent`, the type `sptr` is defined as a pointer to a variable of type `s`, with the intention of referring to the local declaration of `s`. However, the Compiler uses the global declaration of `s`. This produces the Compiler error shown in the comment, because `ptrs^` and `current` are assignment incompatible. To avoid this kind of problem, you could redeclare the type `s` locally before using it in a nested block. The more general solution, however, is to avoid redeclaring identifiers of pointer base types altogether.

Appendix C **Reserved Words and the Character Set**

THIS APPENDIX PROVIDES A COMPLETE LIST of the MPW Pascal reserved words and the character set. ■

Contents

Reserved words

AND	DOWNTO	IF	NIL	PROGRAM	TYPE
ARRAY	ELSE	IMPLEMENTATION	NOT	RECORD	UNIT
BEGIN	END	IN	OF	REPEAT	UNTIL
CASE	FILE	INTERFACE	OR	SET	USES
CONST	FOR	INTRINSIC*	OTHERWISE	STRING	VAR
DIV	FUNCTION	LABEL	PACKED	THEN	WHILE
DO	GOTO	MOD	PROCEDURE	TO	WITH

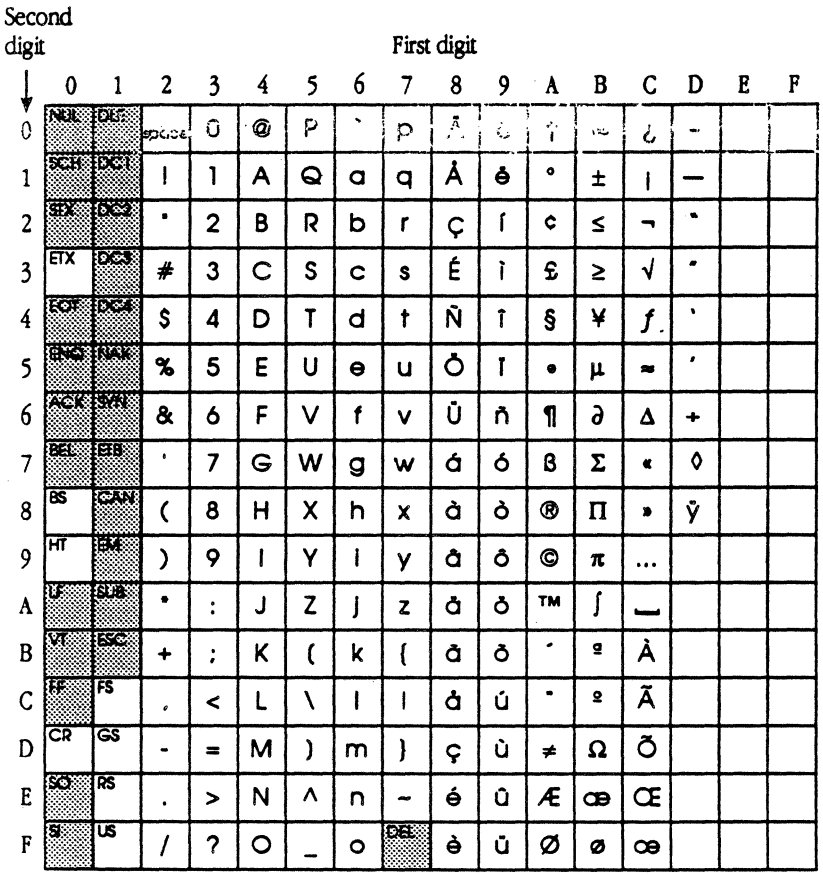
* INTRINSIC is reserved for future use.

Reserved words appear in uppercase letters throughout this book. However, MPW Pascal isn't case sensitive—corresponding uppercase and lowercase letters are equivalent.

The character set

The first two columns of the character set in Figure C-1 are nonprinting ASCII control characters. Codes $D9 through $FF are reserved for future use.

■ **Figure C-1** The character set



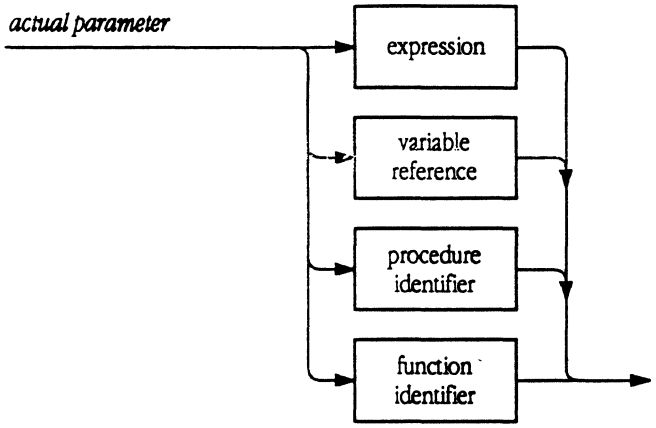
Second digit ↓	First digit 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	space	0	@	P	`	p	Ä	ê	†	≈	¿	-		
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	"		
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√	'		
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	'		
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈	'		
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷		
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	Σ	«	◊		
8	BS	CAN	(	8	H	X	h	x	à	ò	®	Π	»	ÿ		
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…			
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫				
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª	À			
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º	Ã		
D	CR	GS	-	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ			
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ			

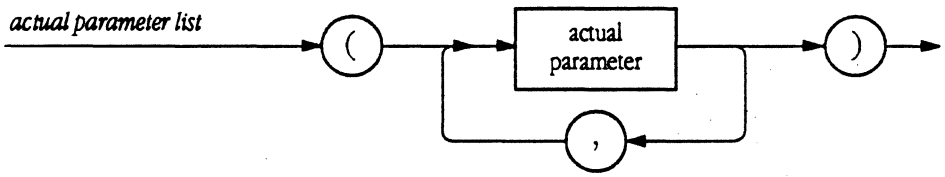
 stands for a nonbreaking space, the same width as a digit.
The shaded characters cannot normally be generated from the Macintosh keyboard or keypad.

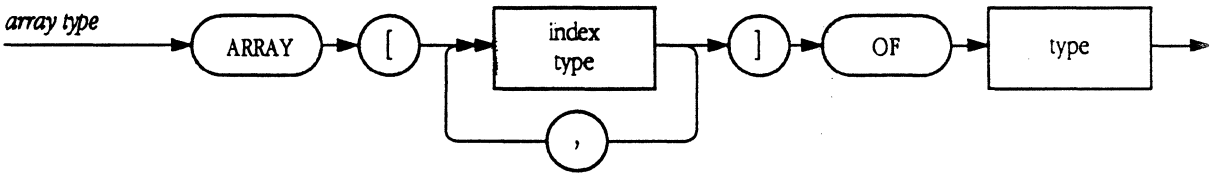
■ The light-shaded characters are not in all fonts.

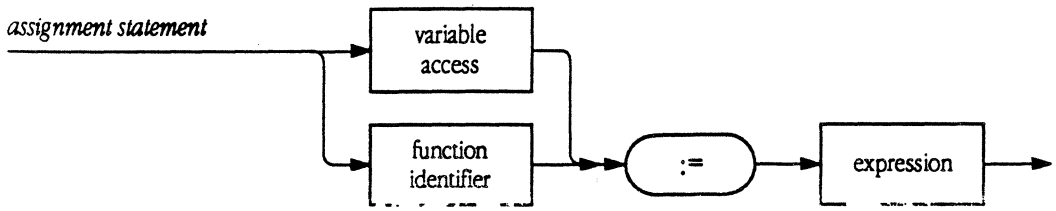
Appendix D **Syntax Summary**

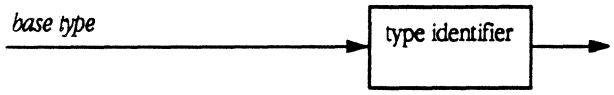
THIS APPENDIX COLLECTS THE SYNTAX DIAGRAMS found in this manual and shows them in alphabetic order. See the Preface for an explanation of them. ■

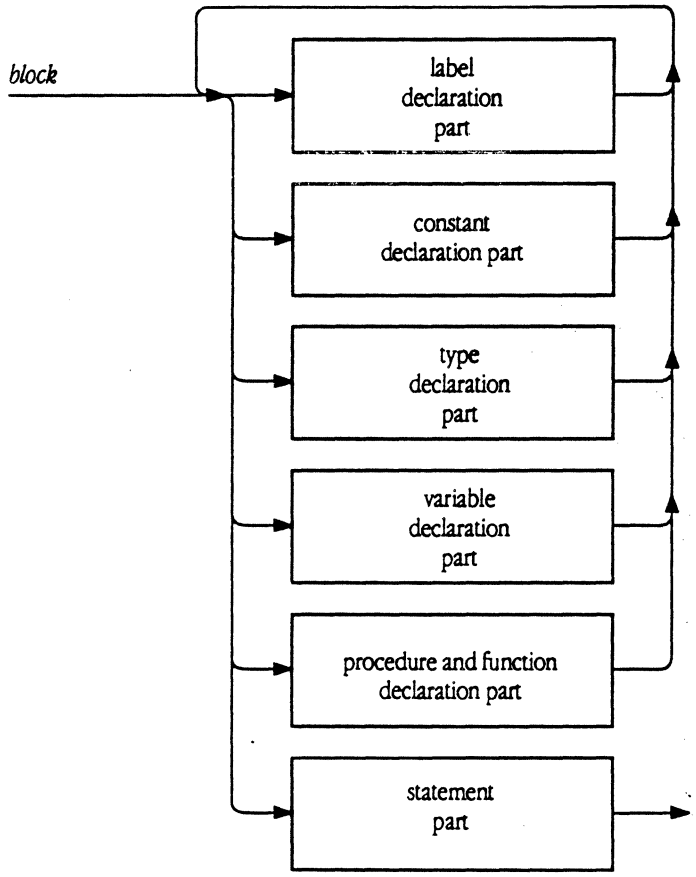

actual parameter
expression
variable reference
procedure identifier
function identifier

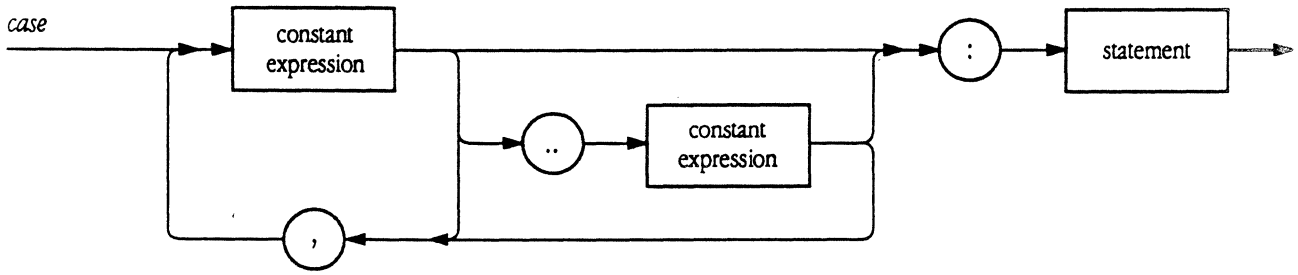

actual parameter list
(
actual parameter
,
)

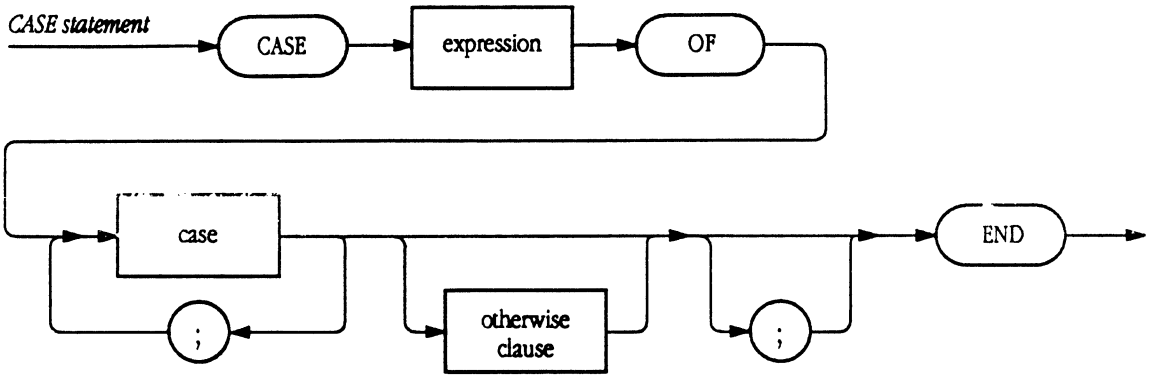

array type
ARRAY
[
index type
,
]
OF
type

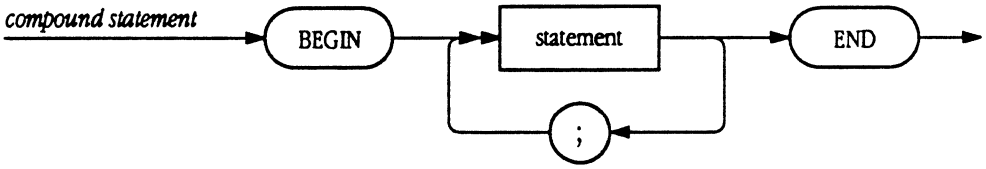

assignment statement
variable access
function identifier
:=
expression

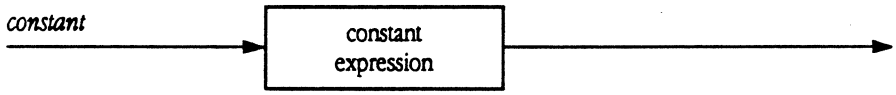

base type
type identifier

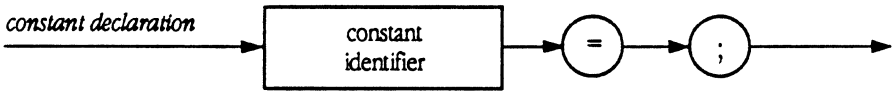

block
label declaration part
constant declaration part
type declaration part
variable declaration part
procedure and function declaration part
statement part

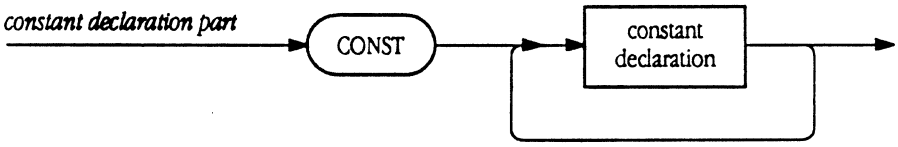

case
constant expression
:
statement
..
constant expression
,

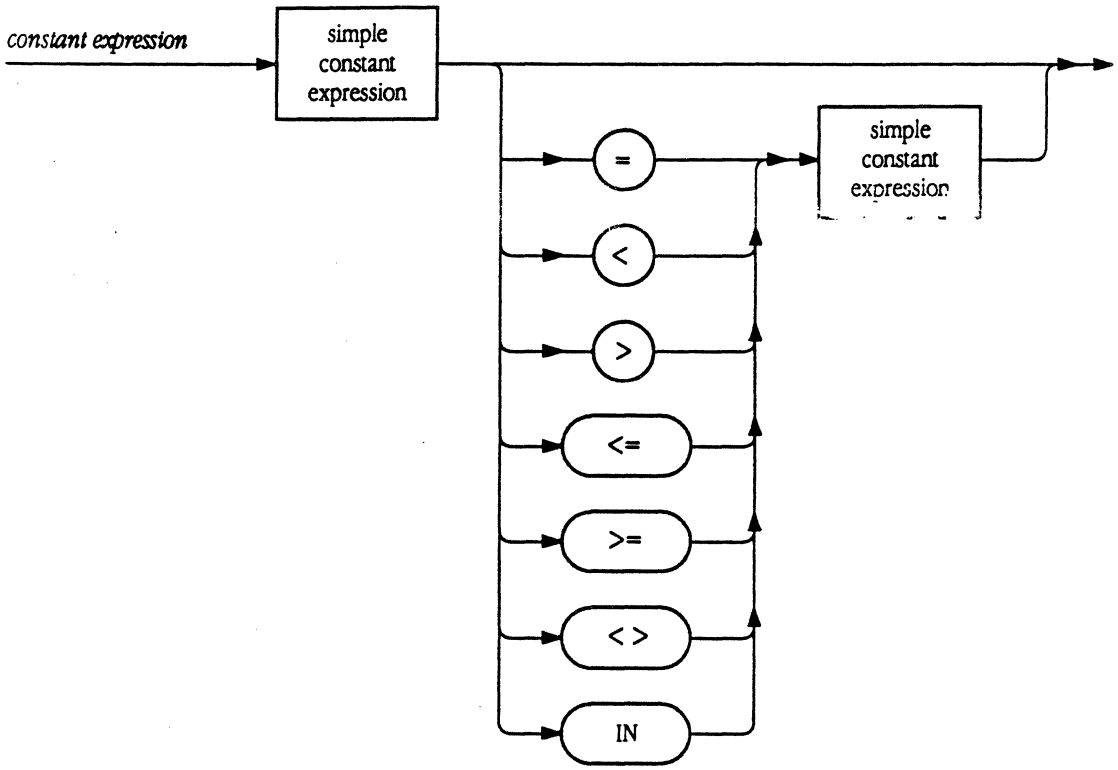

CASE statement
CASE
expression
OF
case
;
otherwise clause
;
END

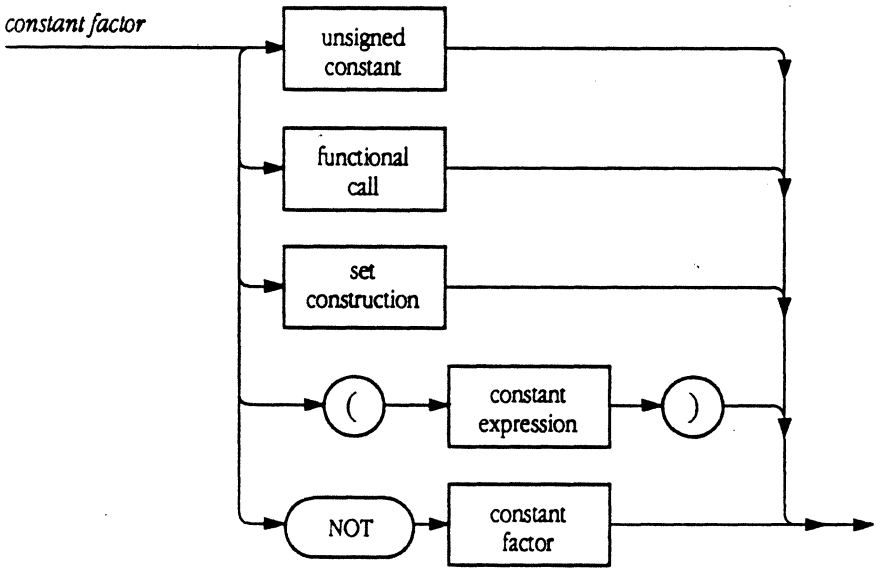

compound statement
BEGIN
statement
;
END

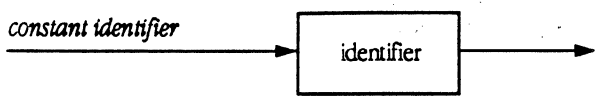

constant
constant expression

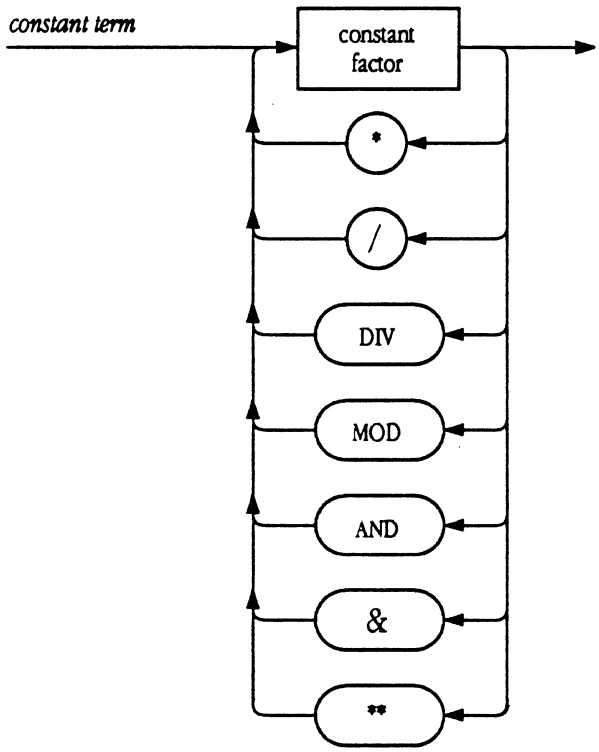

constant declaration
constant identifier
=
;

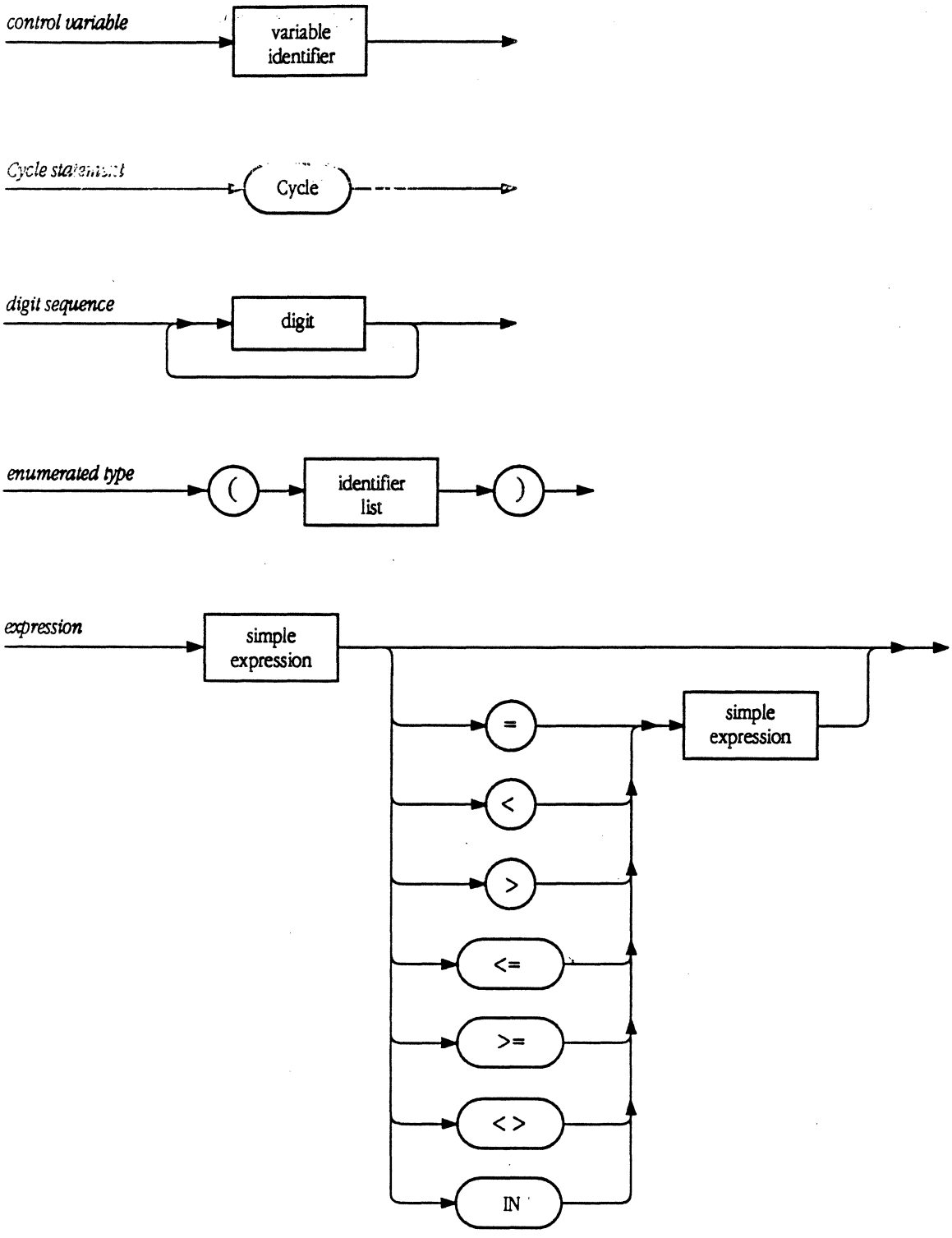

constant declaration part
CONST
constant declaration

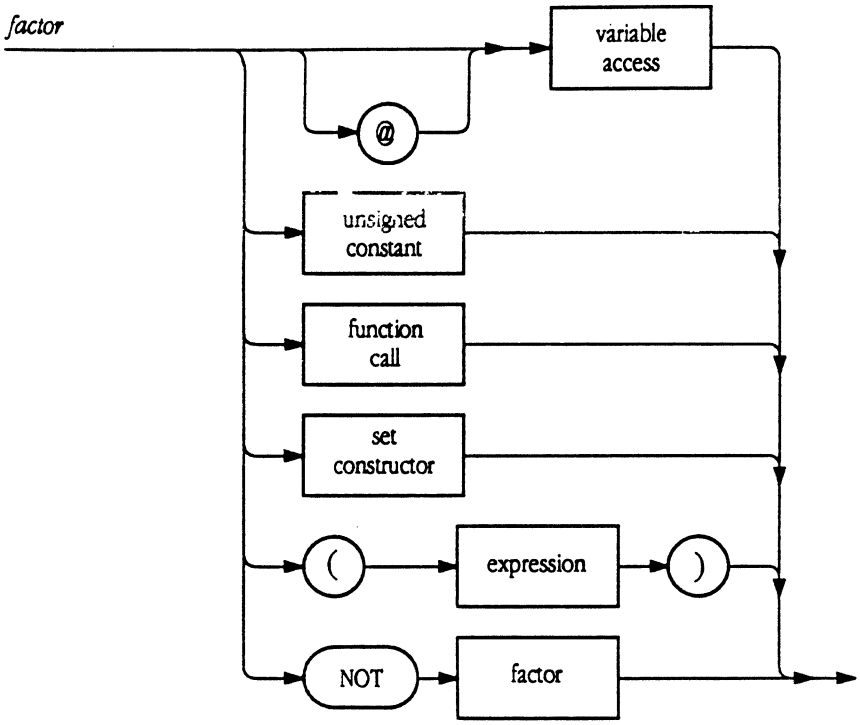

constant expression
simple constant expression
=
<
>
<=
>=
<>
IN
simple constant expression

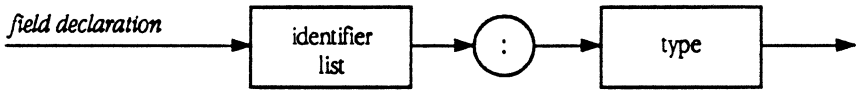

constant factor
unsigned constant
functional call
set construction
(
constant expression
)
NOT
constant factor

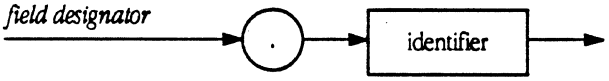

constant identifier
identifier

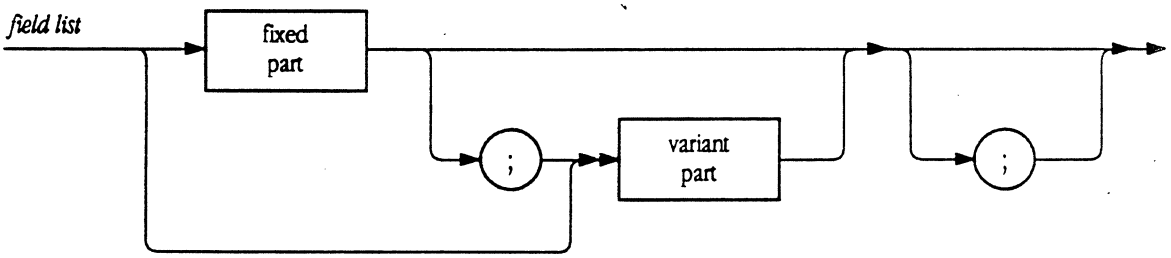

constant term
constant factor
*
/
DIV
MOD
AND
&
**

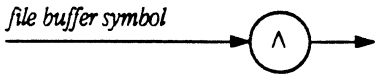

control variable
variable identifier
Cycle statement
Cycle
digit sequence
digit
enumerated type
(
identifier list
)
expression
simple expression
=
<
>
<=
>=
<>
IN
simple expression

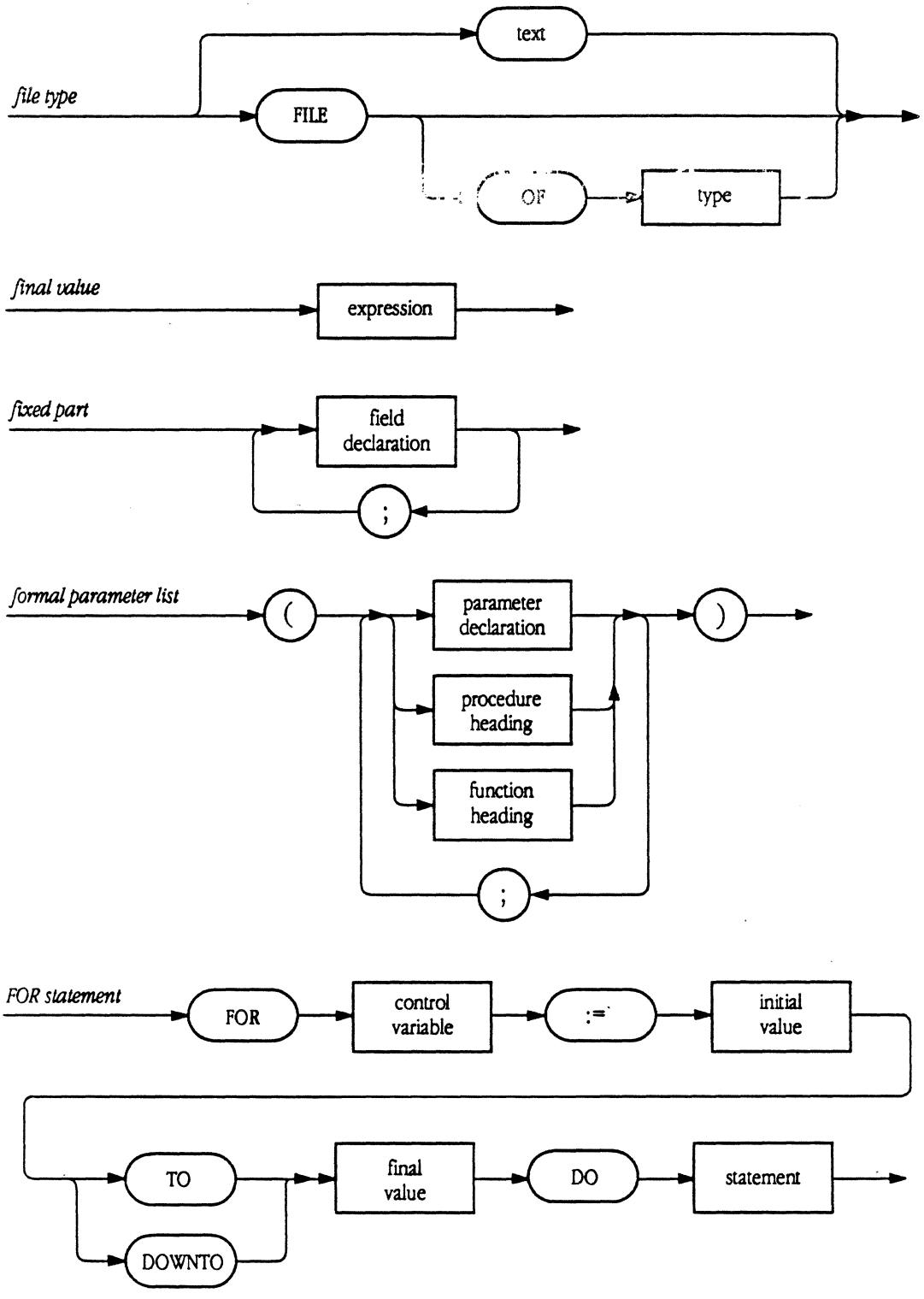

factor
variable access
@
unsigned constant
function call
set constructor
(
expression
)
NOT
factor

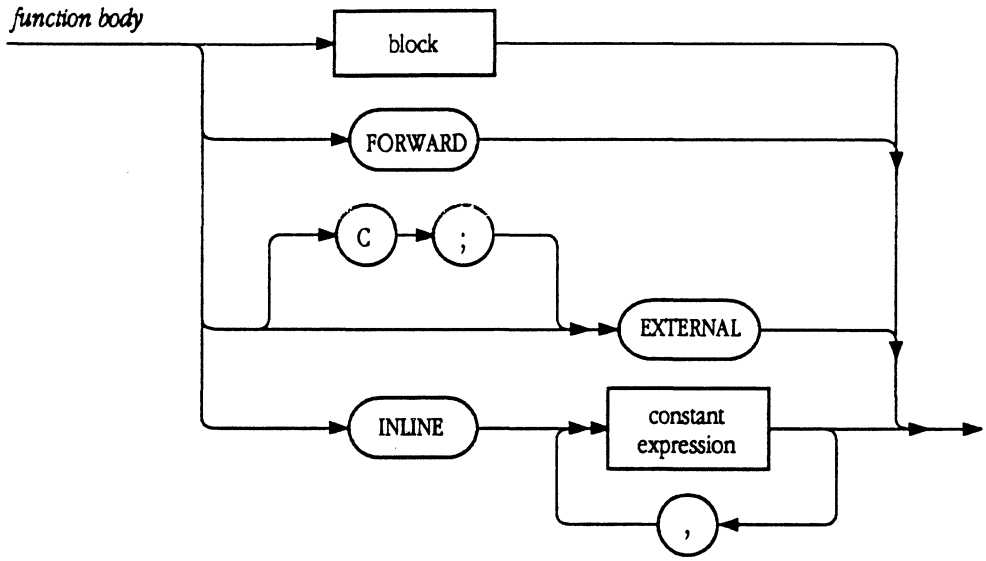

field declaration
identifier list
:
type

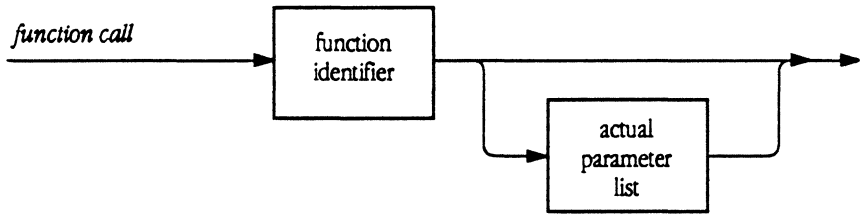

field designator
.
identifier

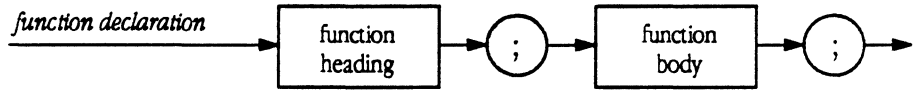

field list
fixed part
;
variant part
;

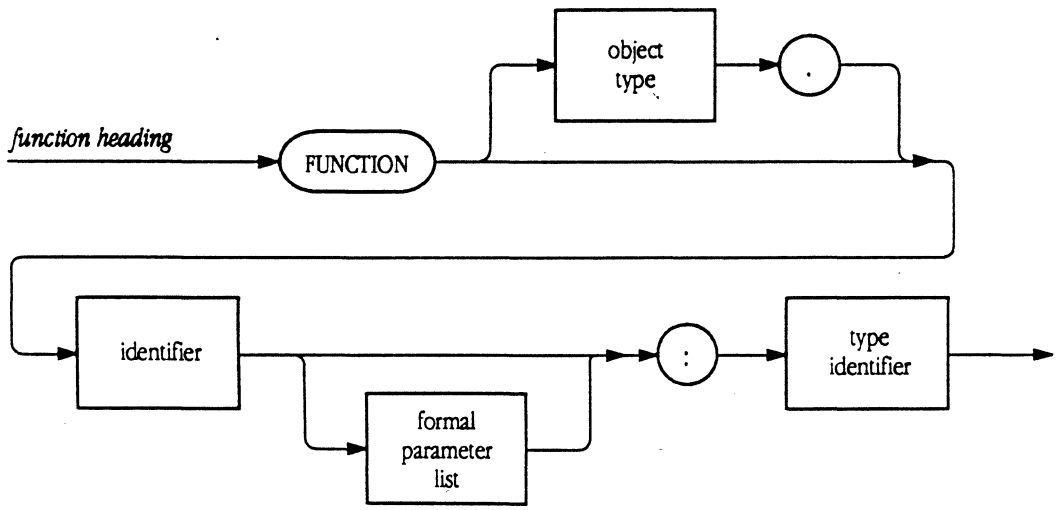

file buffer symbol
^


file type
text
FILE
OF
type
final value
expression
fixed part
field declaration
;
formal parameter list
(
parameter declaration
)
procedure heading
function heading
;
FOR statement
FOR
control variable
:=
initial value
TO
DOWNTO
final value
DO
statement

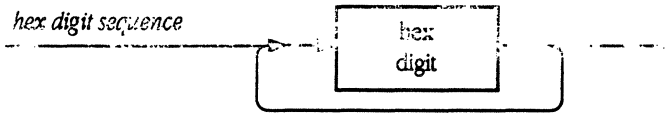

function body
block
FORWARD
C
;
EXTERNAL
INLINE
constant expression
,

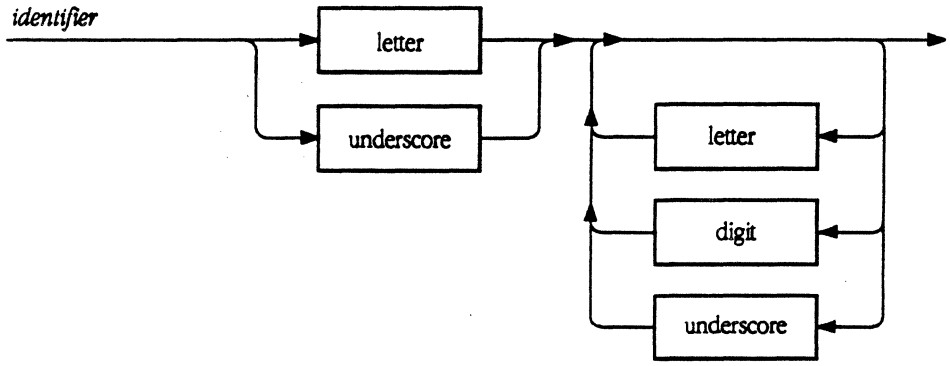

function call
function identifier
actual parameter list

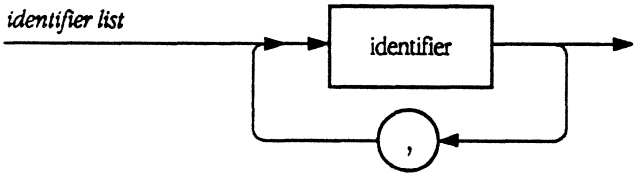

function declaration
function heading
;
function body
;

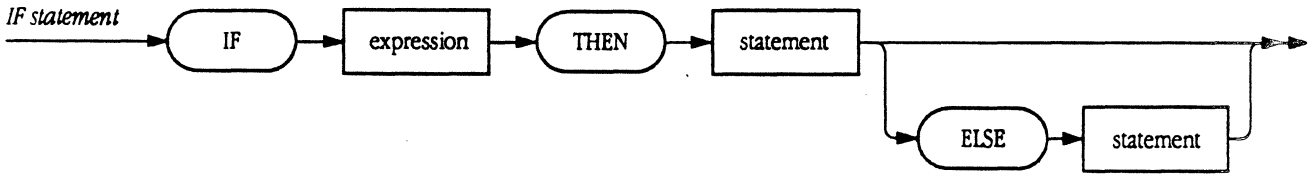

function heading
object type
.
FUNCTION
identifier
formal parameter list
:
type identifier

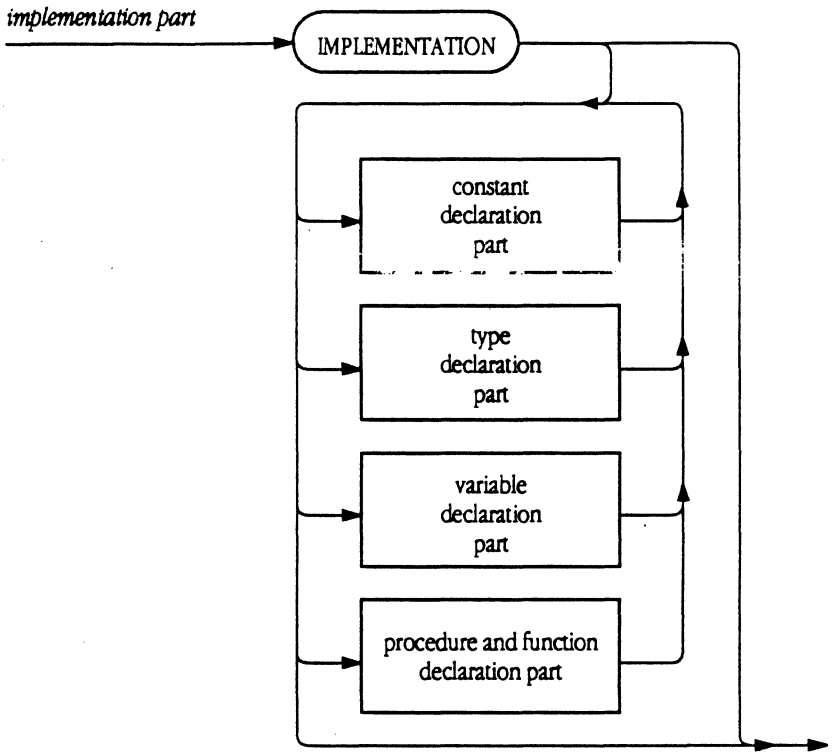

GOTO statement
GOTO
label

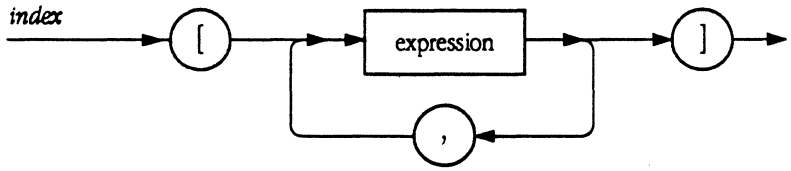

hex digit sequence
hex
digit

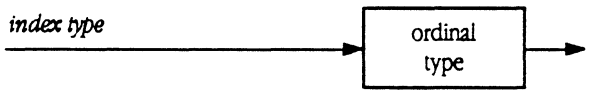

identifier
letter
underscore
letter
digit
underscore

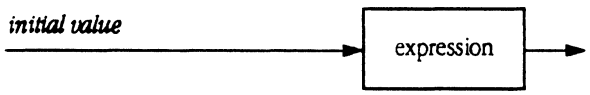

identifier list
identifier
,

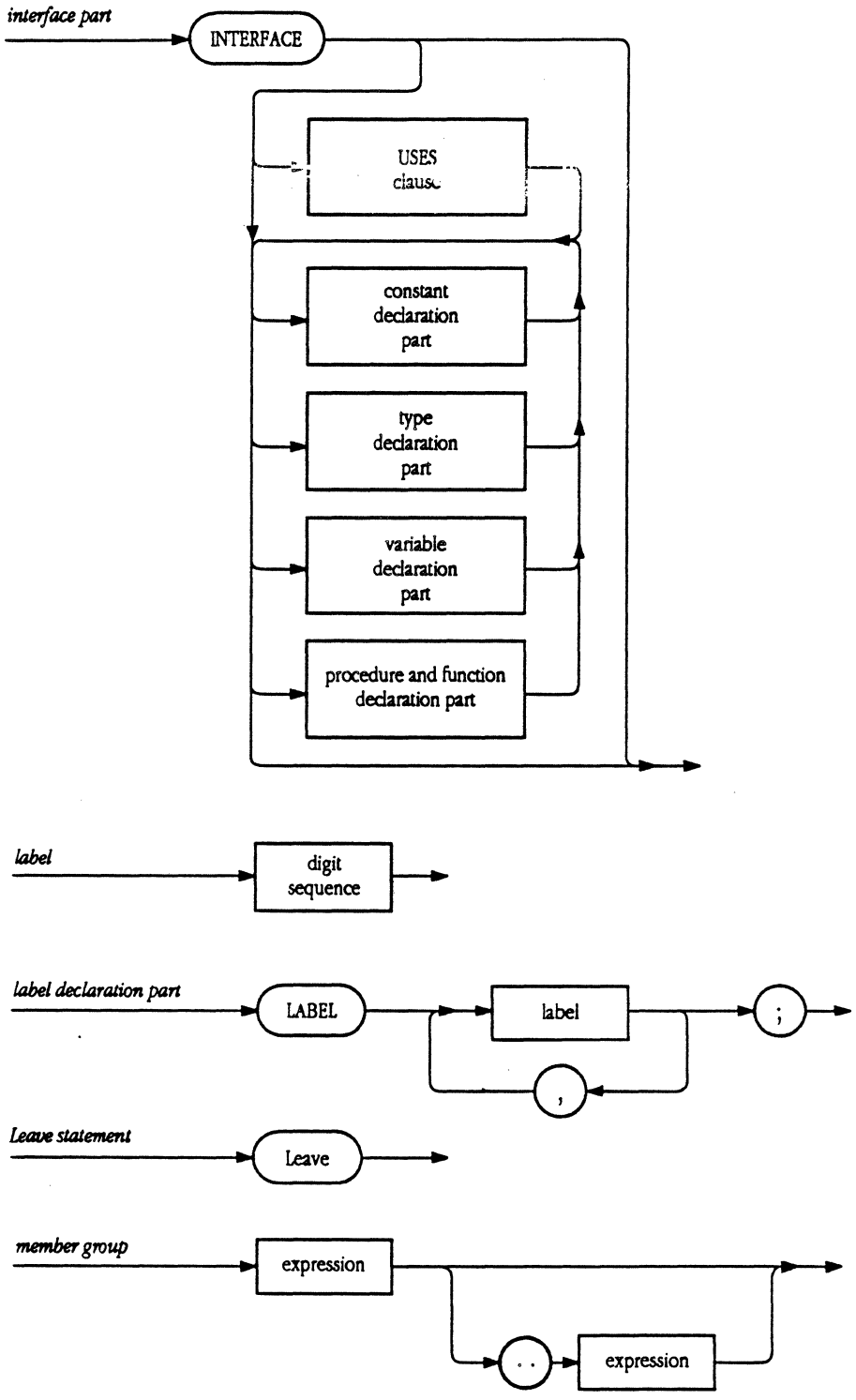

IF statement
IF
expression
THEN
statement
ELSE
statement

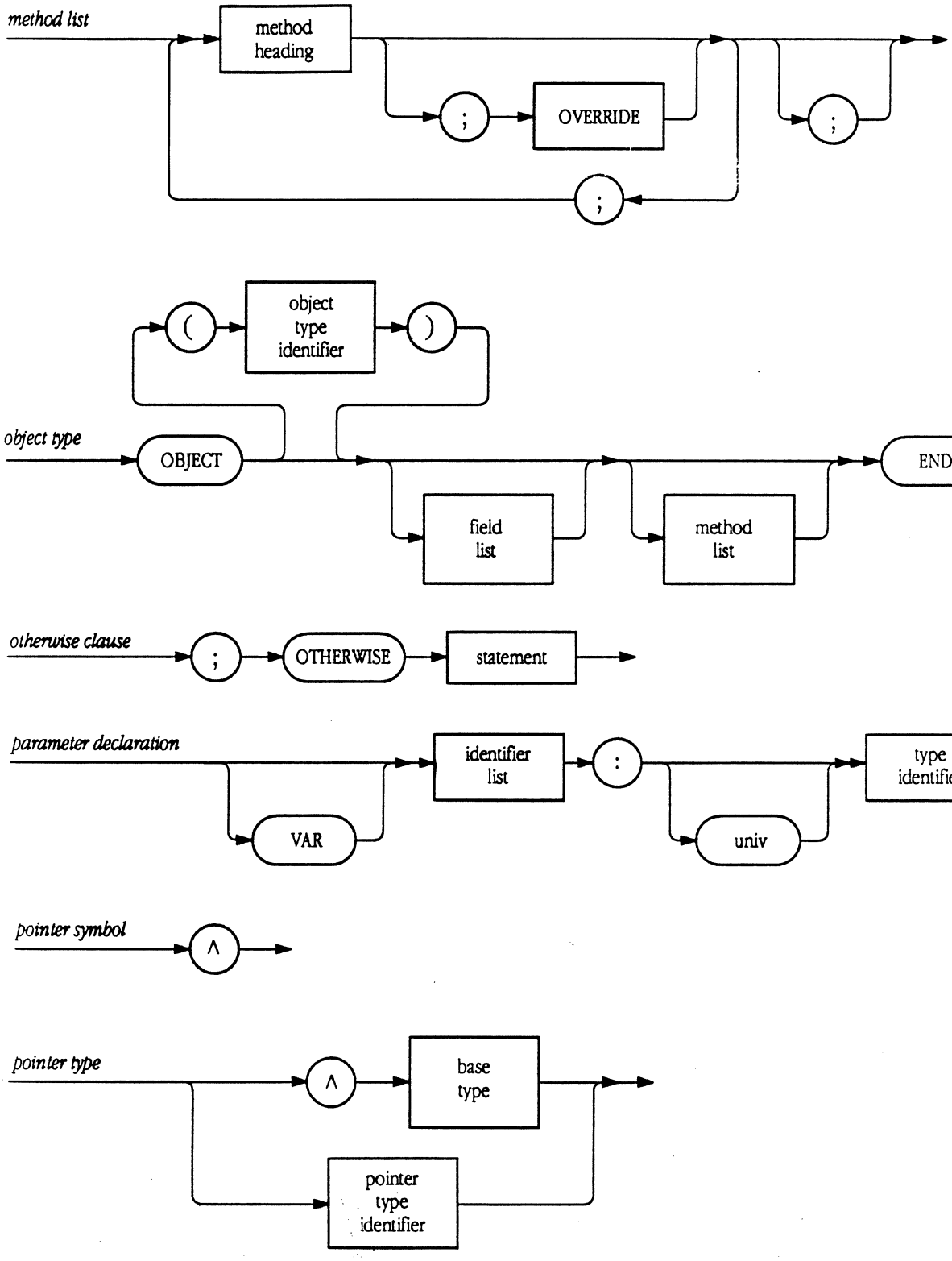

implementation part
IMPLEMENTATION
constant declaration part
type declaration part
variable declaration part
procedure and function declaration part

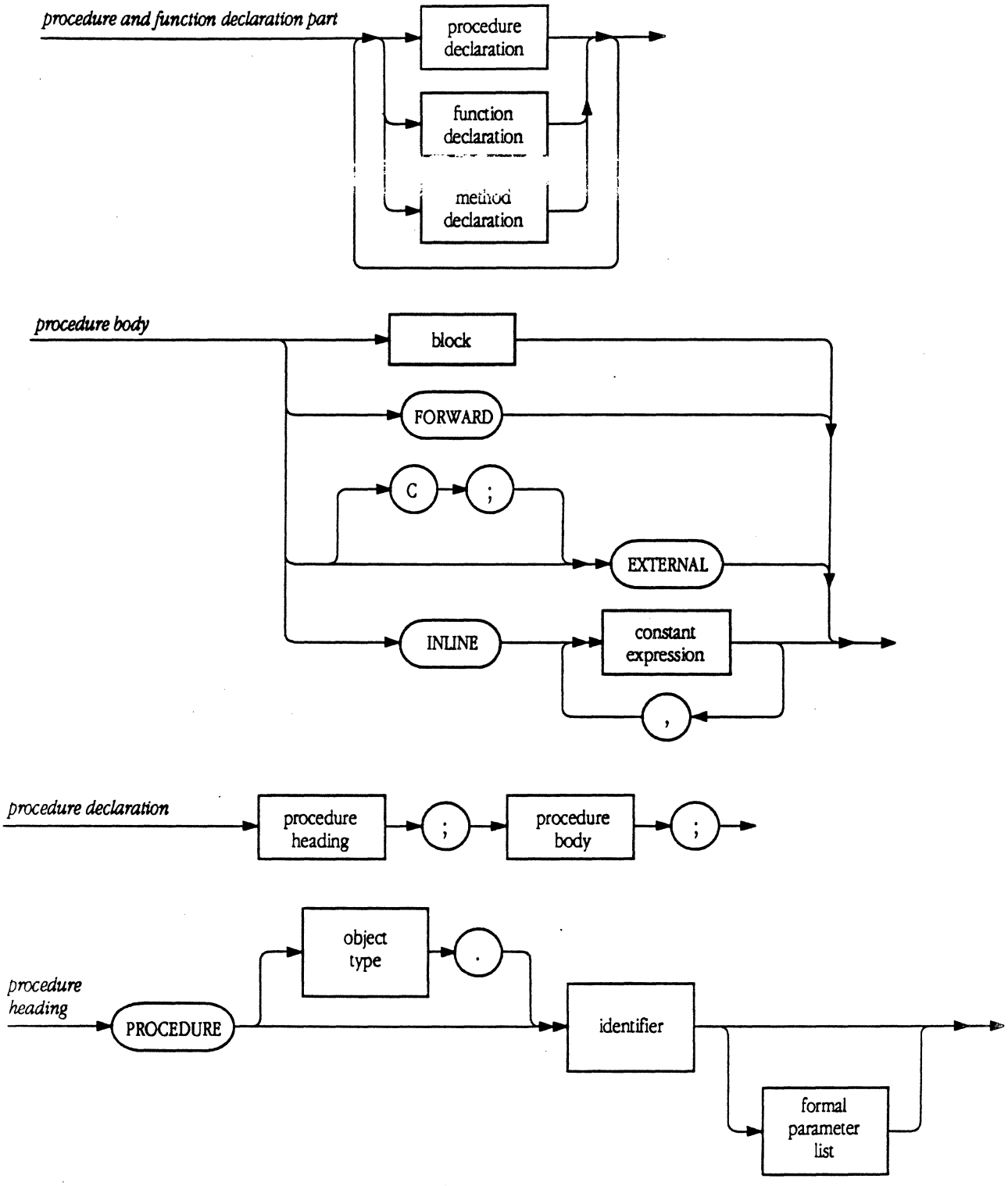

index
[
expression
]
,

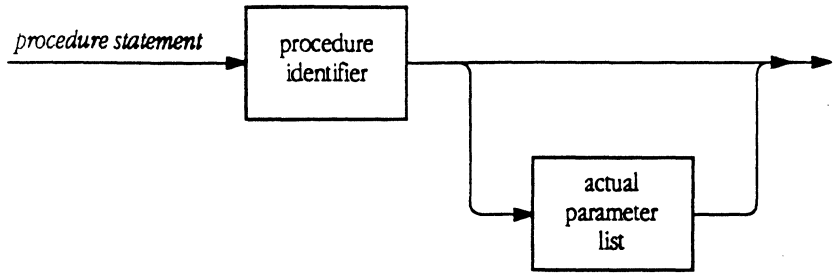

index type
ordinal type

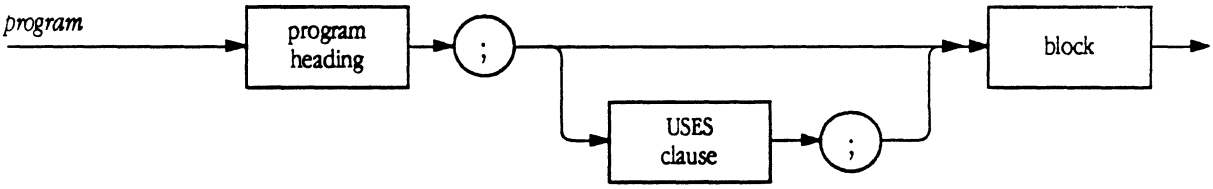

initial value
expression

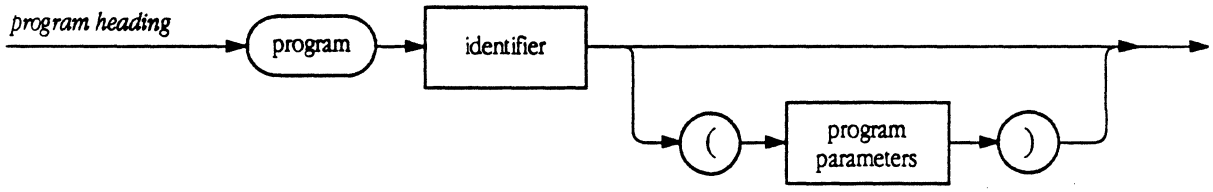

interface part
INTERFACE
USES clause
constant declaration part
type declaration part
variable declaration part
procedure and function declaration part
label
digit sequence
label declaration part
LABEL
label
;
,
Leave statement
Leave
member group
expression
. .
expression

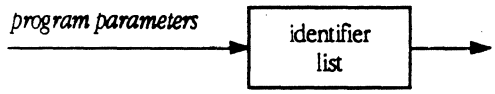

method list
method heading
;
OVERRIDE
;
;
object type
(
object type identifier
)
OBJECT
field list
method list
END
otherwise clause
;
OTHERWISE
statement
parameter declaration
VAR
identifier list
:
univ
type identifier
pointer symbol
^
pointer type
^
base type
pointer type identifier

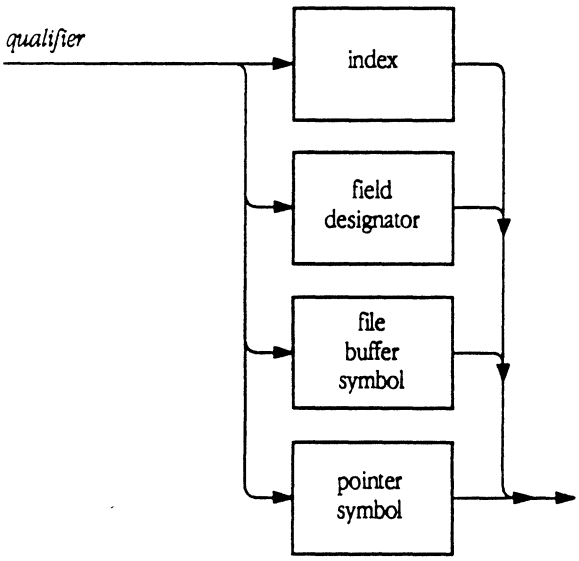

procedure and function declaration part
procedure declaration
function declaration
method declaration
procedure body
block
FORWARD
C
;
EXTERNAL
INLINE
constant expression
,
procedure declaration
procedure heading
;
procedure body
;
procedure heading
PROCEDURE
object type
.
identifier
formal parameter list

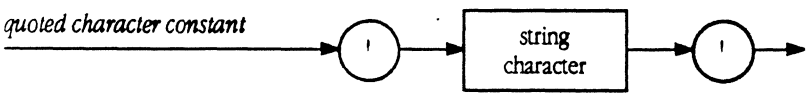

procedure statement
procedure identifier
actual parameter list

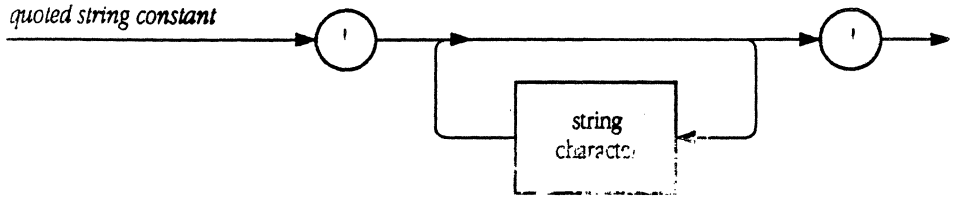

program
program heading
;
block
USES clause
;

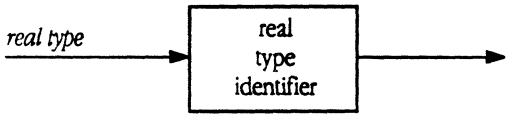

program heading
program
identifier
(
program parameters
)

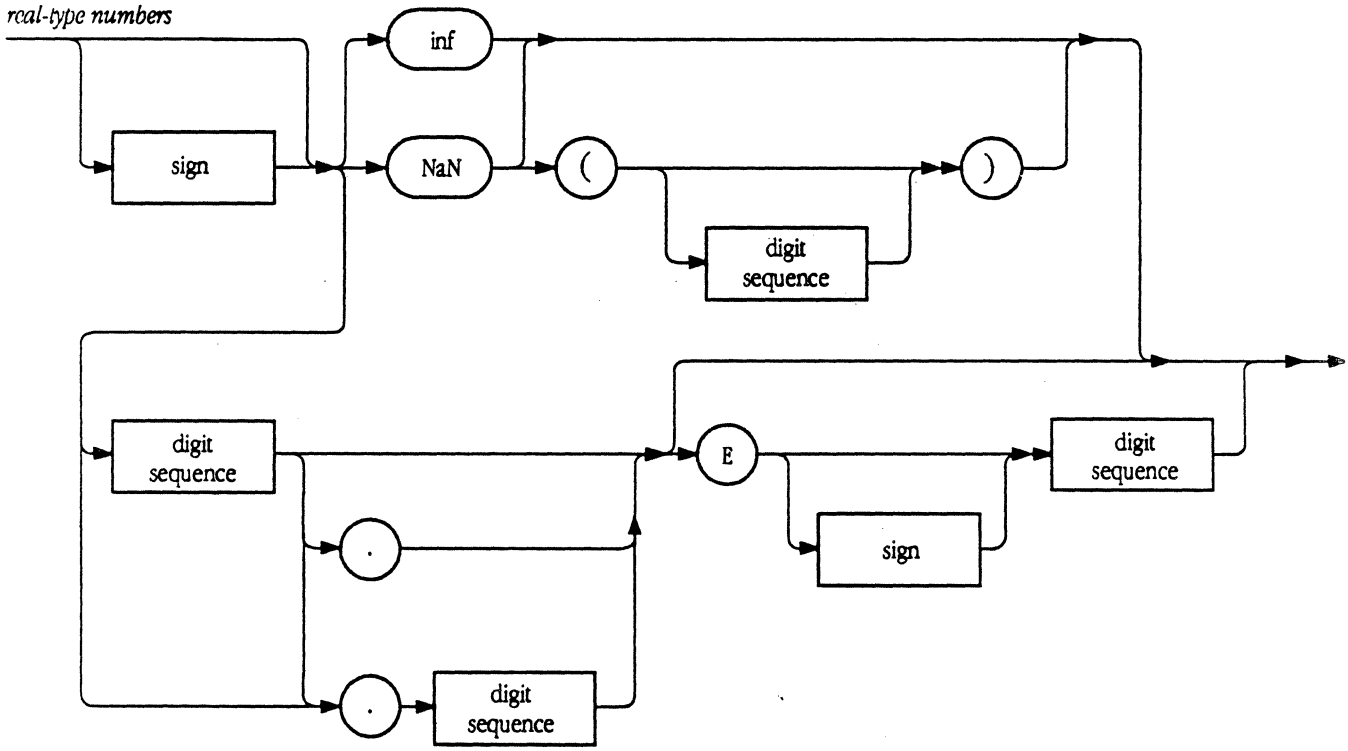

program parameters
identifier list

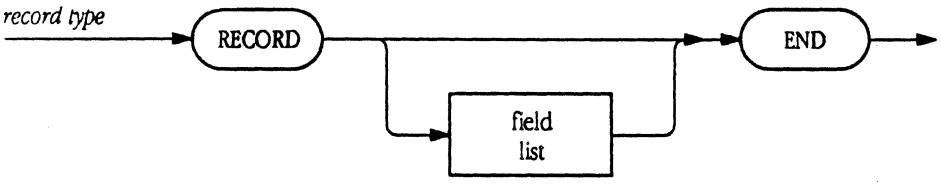

qualifier
index
field designator
file buffer symbol
pointer symbol

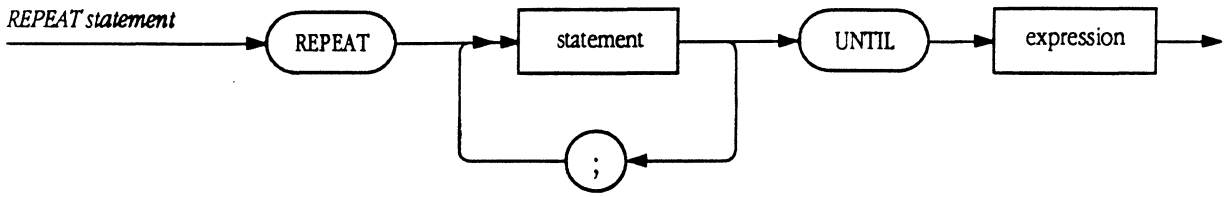

quoted character constant
'
string character
'

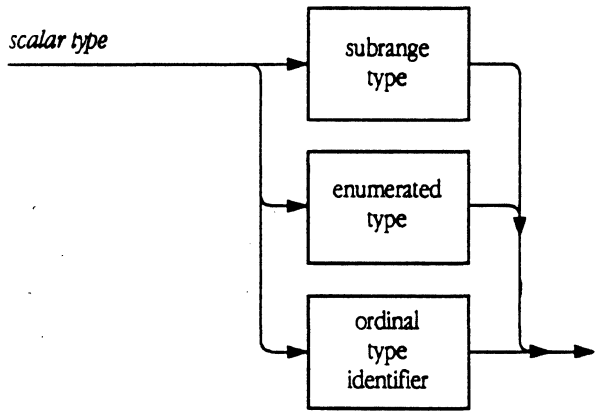

quoted string constant
'
'
string
character

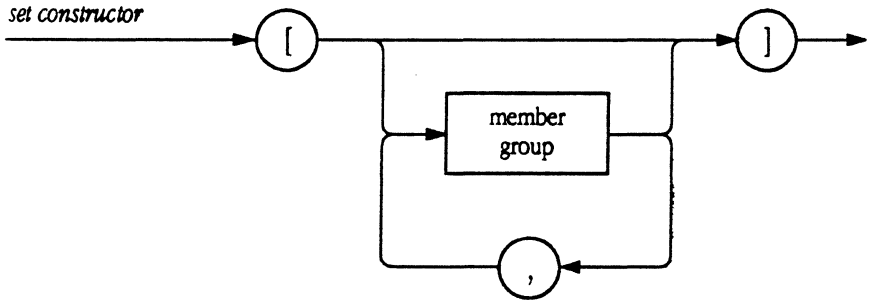

real type
real
type
identifier

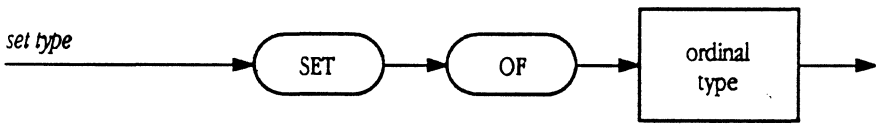

real-type numbers
inf
sign
NaN
(
)
digit
sequence
digit
sequence
E
digit
sequence
.
sign
.
digit
sequence

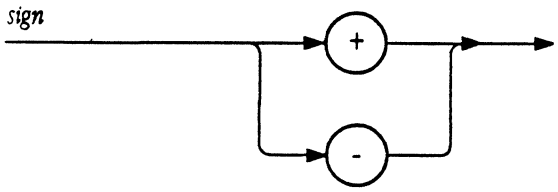

record type
RECORD
END
field
list

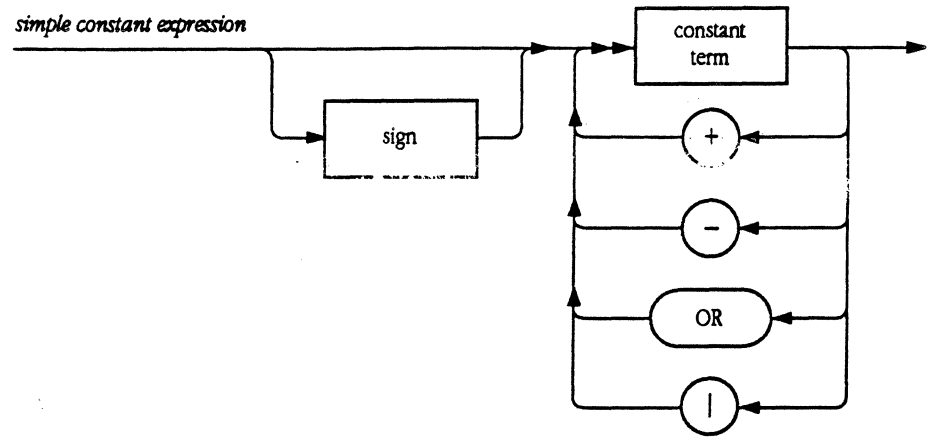

REPEAT statement
REPEAT
statement
UNTIL
expression
;

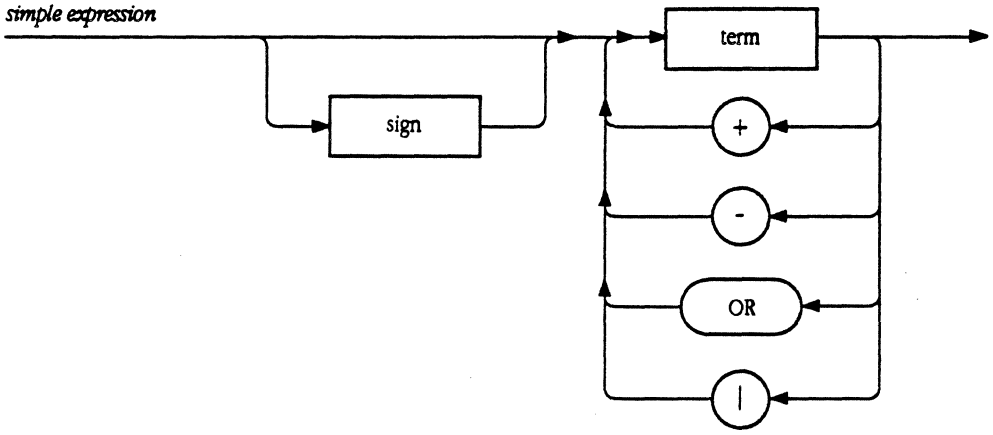

scalar type
subrange type
enumerated type
ordinal type identifier

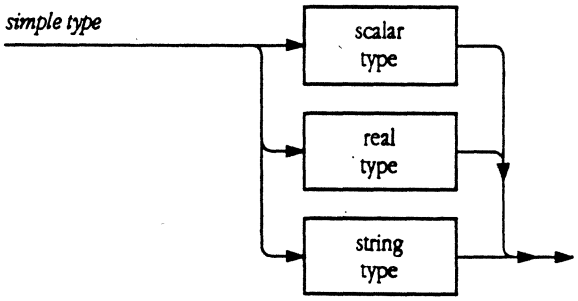

set constructor
[
]
member group
,

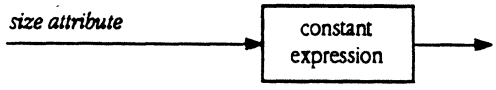

set type
SET
OF
ordinal type

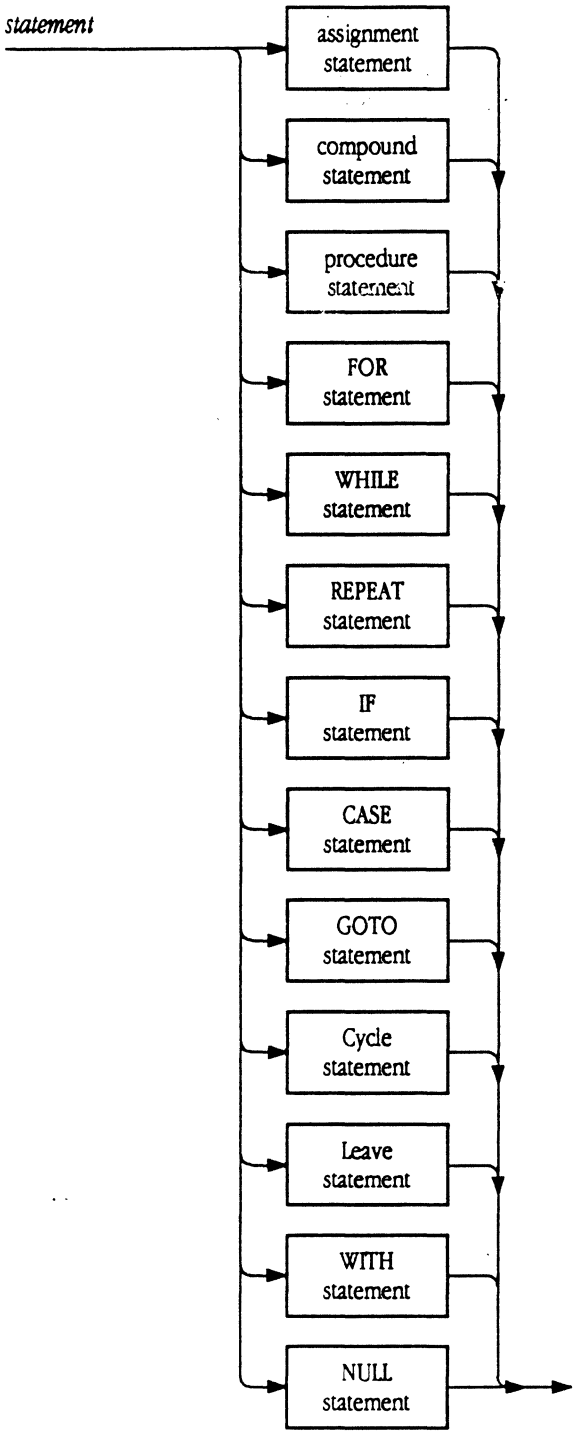

sign
+
-

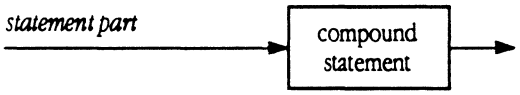

simple constant expression
sign
constant term
+
-
OR
|

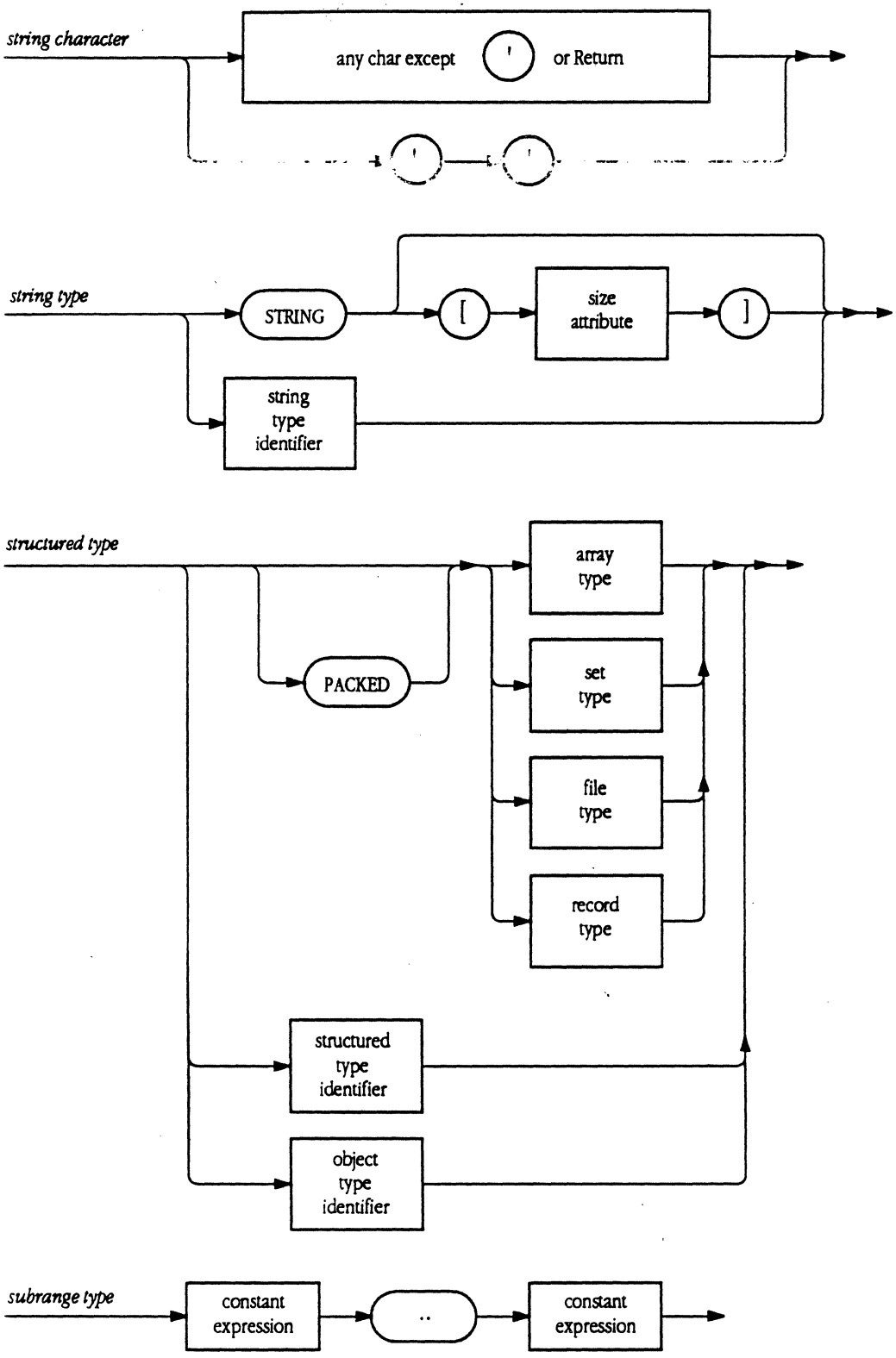

simple expression
sign
term
+
-
OR
|

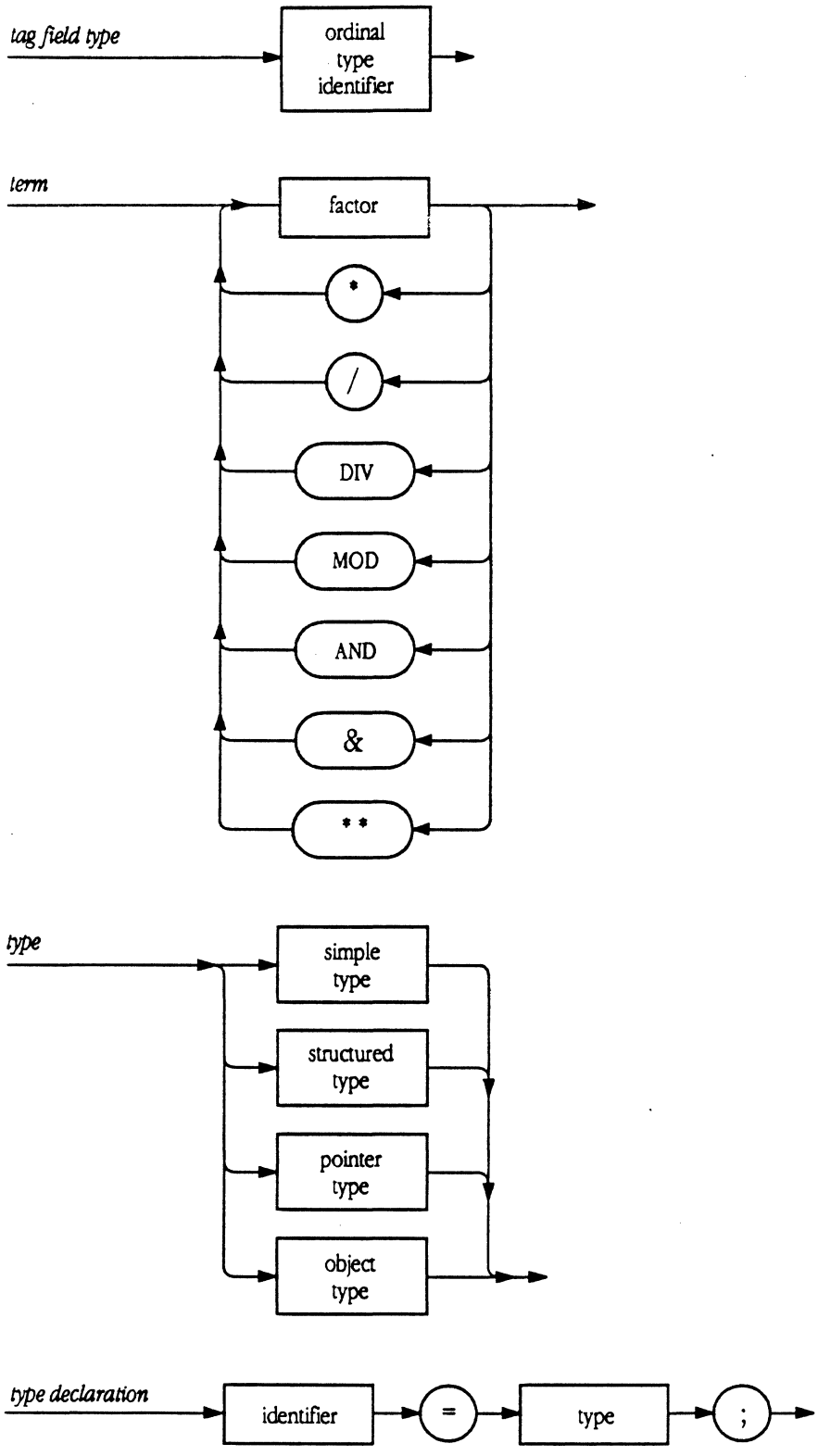

simple type
scalar type
real type
string type

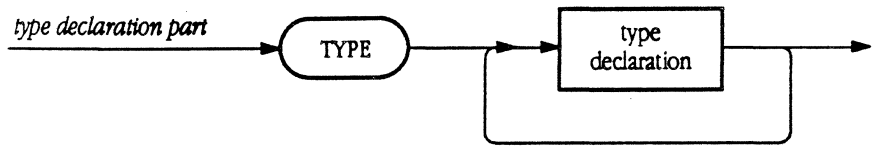

size attribute
constant expression

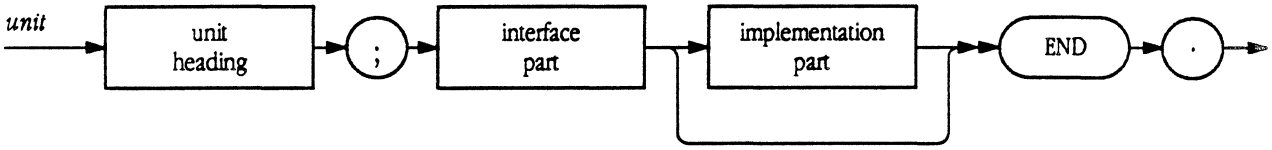

statement
assignment statement
compound statement
procedure statement
FOR statement
WHILE statement
REPEAT statement
IF statement
CASE statement
GOTO statement
Cycle statement
Leave statement
WITH statement
NULL statement

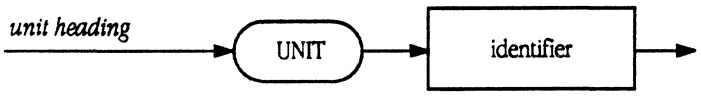

statement part
compound statement

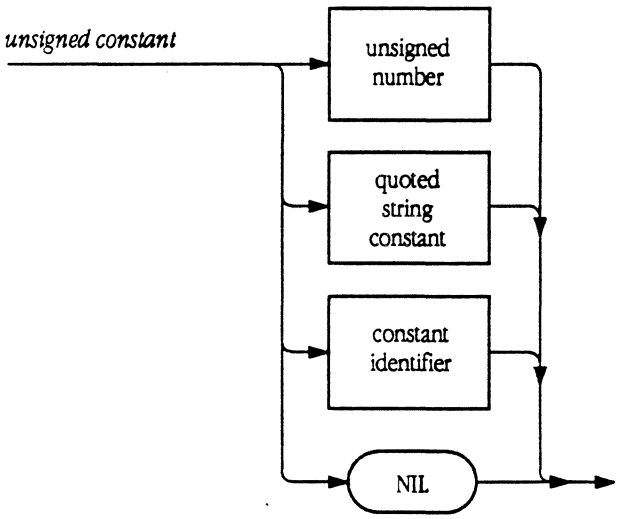

string character
any char except
'
or Return
'
'
string type
STRING
[
size attribute
]
string type identifier
structured type
array type
PACKED
set type
file type
record type
structured type identifier
object type identifier
subrange type
constant expression
..
constant expression

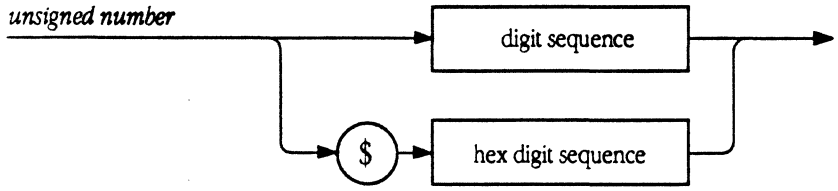

tag field type
ordinal type identifier
term
factor
*
/
DIV
MOD
AND
&
* *
type
simple type
structured type
pointer type
object type
type declaration
identifier
=
type
;

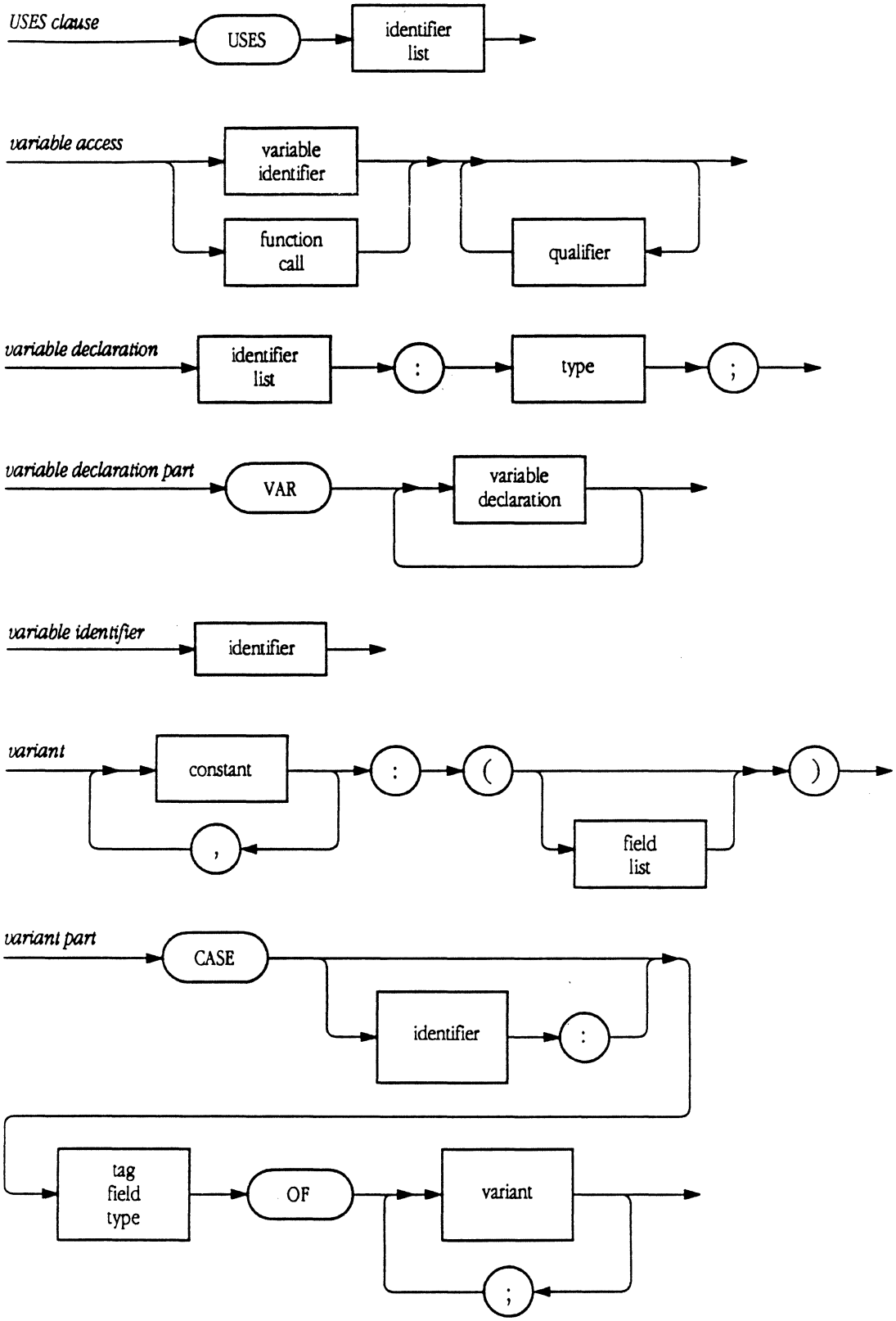

type declaration part
TYPE
type declaration

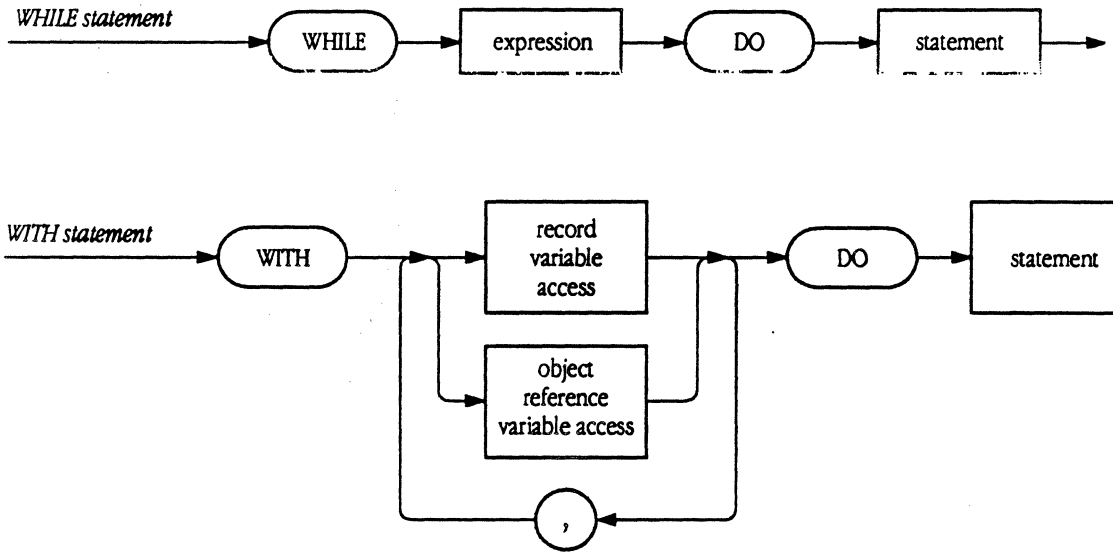

unit
unit heading
;
interface part
implementation part
END
.


unit heading
UNIT
identifier


unsigned constant
unsigned number
quoted string constant
constant identifier
NIL


unsigned number
digit sequence
$
hex digit sequence


USES clause
USES
identifier list
variable access
variable identifier
function call
qualifier
variable declaration
identifier list
:
type
;
variable declaration part
VAR
variable declaration
variable identifier
identifier
variant
constant
:
(
)
,
field list
variant part
CASE
identifier
:
tag field type
OF
variant
;


WHILE statement
WHILE
expression
DO
statement
WITH statement
WITH
record variable access
object reference variable access
,
DO
statement

Appendix E **MPW 3.0 Pascal Files**

THIS APPENDIX CONSISTS OF A LIST OF THE FILES CONTAINED on the MPW 3.0 Pascal disk: MPW 3.0 Pascal.

◆ *Note:* For the latest list of MPW Pascal files, consult the MPW 3.0 Pascal release letter. ■

Contents

Pascal compiler and tools

Pascal	Pascal Compiler
PasMat	Pascal print formatter ("pretty printer")
PasRef	Pascal cross-referencer

PExamples folder

Fstubs.a	Dummy library routines that override those not used by MPW tool
Instructions	Instructions for building example program
MakeFile	Makefile for Sample program
Memory.p	Sample MPW tool
Memory.r	Resource description file for Memory.p
ResEd.p	Routines for extending ResEdit
ResEd68K.a	Routines for extending ResEdit
ResEqual.p	Sample MPW tool
ResEqual.r	Resource description file for ResEqual.p
ResXXXXEd.p	Sample resource editor
Sample.p	Sample application
Sample.r	Resource description file for Sample.p
TestPerf.p	Sample Performance tool

PInterfaces folder

AppleTalk.p	AppleTalk interface
Controls.p	Control Manager interface
CursorCtl.p	MPW cursor-control interface
Desk.p	Desk manager interface
DeskBus.p	Apple Desktop Bus Manager interface
Devices.p	Device Manager interface
Dialogs.p	Dialog Manager interface
DisAsmLookup.p	SADE and MacsBug symbols
DiskInit.p	Disk Initialization Package interfaces
Disks.p	Disk Driver interfaces

ErrMgr.p	MPW Error Manager interface
Errors.p	Error file
Events.p	Event Manager interfaces
Files.p	File Manager interfaces
FixMath.p	Interface for fixed-point mathematics routines
Fonts.p	Font Manager interface
Graf3D.p	3-D graphics interface
HyperXCmd.p	HyperCard XCmd interfaces
IntEnv.p	Integrated Environment (MPW tool) interface
Lists.p	List Manager interface
MacPrint.p	Printing interface—`includes` Printing.p
Memory.p	Memory Manager interface
MemTypes.p	Common types—`includes` Types.p
Menus.p	Menu Manager interfaces
Notification.p	Notification Manager
ObjIntf.p	Object Pascal support
OSEvents.p	OS Event Manager interfaces
OSIntf.p	Operating system interface—`includes` OSUtils.p, Events.p, Files.p, Devices.p, DeskBus.p, DiskInit.p, Disks.p, Errors.p, Memory.p, OSEvents.p, Retrace.p, Segload.p, Serial.p, Shutdown.p, Slots.p, Sound.p, Start.p, and Timer.p
OSUtils.p	Operating System Utilities interfaces
Packages.p	Package Manager interfaces
PackIntf.p	Packages interface—`includes` Packages.p
PaletteMgr.p	Palette Manager—`includes` Palettes.p
Palettes.p	Palette Manager interfaces
PasLibIntf.p	Pascal Library interface
Perf.p	Pascal Performance tools
Picker.p	Color Picker Manager interfaces
PickerIntf.p	Color Picker Manager—`includes` Picker.p
Printing.p	Alternate printing interface
PrintTraps.p	Preferred printing interface
QuickDraw.p	QuickDraw interface
Resources.p	Resources Manager interfaces
Retrace.p	Vertical Retrace Manager interfaces
ROMDefs.p	ROM definitions

SANE.p	SANE numerics interface
Scrap.p	Scrap Manager interfaces
Script.p	International writing interface
SCSI.p	SCSI Manager interfaces
SCSIIntf.p	SCSI interface—`includes` SCSI.p
SegLoad.p	Segment Loader interfaces
Serial.p	Serial Driver interfaces
Shutdown.p	Shut Down Manager interfaces
Signal.p	Signal-handling interface (talks with C-style interface)
Slots.p	Slot Manager interfaces
Sound.p	Updated Sound Manager interface
Start.p	Start Manager interfaces
Strings.p	String conversion utilities
SysEqu.p	Low-memory globals
TextEdit.p	Text Edit interfaces
Timer.p	Time Manager interfaces
ToolIntf.p	Macintosh toolbox interface—`includes` ToolUtils.p, Events.p, Controls.p, Desk.p, Windows.p, TextEdit.p, Dialogs.p, Fonts.p, Lists.p, Menus.p, Resources.p, Scrap.p
ToolUtils.p	Toolbox Utilities interfaces
Traps.p	Trap codes
Types.p	Common types
Video.p	Video interface
VideoIntf.p	Video interface—`includes` Video.p
Windows.p	Window Manager interface

PLibraries folder

PasLib.o	Pascal language library, including built-ins and I/O
SANELib.o	SANE numerics library
SANELib881.o	SANE numerics library for use with `-MC68881` Compiler option

Appendix F **Pascal and C Calling Conventions**

THIS APPENDIX DESCRIBES THE TREATMENT of different kinds of parameter and function results by the Pascal Compiler. It covers all the basic data types and discusses C interfacing. ■

Contents

External calling conventions

This section describes the treatment of parameters, function results, and register conventions.

Parameters

Parameters are evaluated from left to right and are pushed onto the stack in that order as they are evaluated. The called procedure is responsible for removing the parameters from the stack. All VAR parameters are passed as pointers to the actual storage location. Note that in cases of byte-wide parameters, the pointer may have an odd value.

Non-VAR parameters are passed in the following ways, depending upon the type of the parameter. Values of type `boolean`, elements of an enumerated type with fewer than 128 elements, and subranges within the range –128..127 are passed as signed byte values. (They are pushed as bytes; the 68000 allocates two bytes for each byte on the stack.) The called procedure expects `boolean` parameters to be in the range 0..1.

Values of types `integer` and `char` and all other enumerations and subranges are passed as signed word values. Pascal `char` values are expected to be in the range 0..255; the upper half of this range is used for special characters. Pointers and `longint` values are passed as signed 32-bit values.

Real type parameters

Values of types `real`, `double`, `comp`, and `extended` are passed as pointers to `extended` values. The Compiler does this in a reentrant way by allocating a temporary location in the caller's activation record, converting the parameter value to an `extended` value in this location and passing a pointer to this location. The called procedure then allocates a local location of the declared type and converts the `extended` value, using the pointer, into the location and type.

Structured type parameters

Arrays, strings, and records whose size is less than or equal to four bytes are passed by pushing their value (either a word or a long) onto the stack. Larger arrays, strings, and records (as well as `extended` values, as mentioned above) are passed as a pointer to the value; for reentry purposes, the Compiler emits code in the called procedure to copy the value to a local storage location.

Sets are passed by rounding the set size up to the next whole word, if necessary, then pushing the set value so that the lowest-order word is pushed last. In the case of a byte-width set, the called procedure will only access the low-order half of the word pushed.

■ **Table F-1** Parameter passing conventions

Parameter type	Pascal caller	Pascal receiver
`boolean`	Pushes byte: range 0..1	Accesses byte: range 0..1
enumeration: range 0..127	Pushes byte: range 0..127	Accesses byte: range 0..127
enumeration range 0..255	Pushes word: range 0..255	Accesses word: range 0..255
enumeration: range 0..32767	Pushes word: range 0..32767	Accesses word: range 0..32767
`char`	Pushes word: range 0..255	Accesses word: range 0..255
subrange: range –128..127	Pushes byte: range –128..127	Accesses byte: range –128..127
subrange: range –32768..32767	Pushes word: range –32768..32767	Accesses word: range –32768..32767
`integer`	Pushes word: range –32768..32767	Accesses word: range –32768..32767
`longint`	Pushes long	Accesses signed long value
pointer	Pushes long	Accesses long
`real`	Converts to `extended`, pushes address of `extended`	Converts `extended` on stack to local `real`, accesses local value

(Continued)

■ **Table F-1** (Continued) Parameter passing conventions

Parameter type	Pascal caller	Pascal receiver
`double`	Converts to `extended`, pushes address of `extended`	Converts `extended` on stack to local `double`, accesses local value
`comp`	Converts to `extended`, pushes address of `extended`	Converts `extended` on stack to local `comp`, accesses local value
`extended`	Pushes address of `extended`	Copies `extended` to local `extended`, accesses local value
`ARRAY`, `RECORD`, `STRING` ≤ four bytes	Pushes value (word or long)	Accesses value (word or long)
`ARRAY`, `RECORD`, `STRING` > four bytes	Pushes address of value	Copies value to local, accesses local
`SET`	Pushes set value rounded to whole number of words	Accesses value on stack (*Note:* Use word or long for those sizes; accesses low-order half of word for byte-size set.)

Function results

Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the parameters are pushed. The caller is responsible for removing the result from the stack after the call.

For types `boolean`, `char`, and `integer`, and enumerated and subrange types, the caller allocates a word on the stack to make space for the function result. Values of type `boolean`, enumerated types with fewer than 128 elements, and subranges within the range –128..127 are returned as signed byte values. The value goes in the low-address byte, which is the most significant byte of the word. The calling procedure expects `boolean` results to be in the range 0..1.

`Integer` and `char` values and all enumerated and subrange types not covered above are returned as signed word values. Pascal `char` values are expected to be in the range 0..255; the upper half of this range is used for special characters.

For pointers, `longint`, and the real types, the caller allocates a long on the stack to make space for the function result. Pointers and `longint` values are returned as signed 32-bit values. Values of type `real` are returned as 32-bit real values. For `double`, `comp`, and `extended` types, and for sets, arrays, strings, and records greater than four bytes in size, the caller pushes a pointer to a temporary location.

For one-byte sets and for arrays, strings, and records whose size is one word, the caller allocates a word on the stack. For sets, arrays, strings, and records whose size is two words, the caller allocates a long word on the stack. One-byte sets are returned as a byte value. Sets, arrays, strings, and records whose sizes are one or two words are returned as either a word or a long word.

■ **Table F-2** Function result passing conventions

Result type	Pascal caller	Pascal receiver	After the call
`boolean`	Allocates word	Returns byte value: range 0..1	Pops byte
enumeration: range 0..127	Allocates word	Returns byte value: range 0..127	Pops byte
enumeration: range 0..32767	Allocates word	Returns word value: range 0..32767	Pops word
`char`	Allocates word	Returns word value: range 0..255	Pops word
subrange: range –128..127	Allocates word	Returns byte value: range –128..127	Pops byte
subrange: range –32768..32767	Allocates word	Returns word value: range –32768..32767	Pops word
`integer`	Allocates word	Returns word value: range –32768..32767	Pops word
`longint`	Allocates long word	Returns long word value: range—signed 32 bits	Pops long word
`real`	Allocates long word	Returns `real` value	Pops real value
`double`	Pushes address of `double` temporary	Puts `double` result in temporary	Pops temporary address, accesses temporary value

(Continued)

■ **Table F-2** (Continued) Function result passing conventions

Result type	Pascal caller	Pascal receiver	After the call
`comp`	Pushes address of `comp` temporary	Puts `double` result in temporary	Pops temporary address, accesses temporary value
`extended`	Pushes address of `extended` temporary	Puts `extended` result in temporary	Pops temporary address, accesses temporary value
`ARRAY`, `STRING`, `RECORD` ≤ 4 bytes	Allocates word or long word	Returns word or long word	Pops word or long word
`ARRAY`, `STRING`, `RECORD` > 4 bytes	Pushes address of temporary	Puts result in temporary	Pops temporary address, accesses temporary value
`SET`: one byte	Allocates word	Returns byte value of result	Pops byte
`SET`: one word	Allocates word	Returns word value of result	Pops word
`SET`: two words	Allocates	Returns long word	Pops long word
`SET` > two words	Pushes address of temporary	Puts result in temporary	Pops temporary address, accesses temporary value

◆ *Note:* Pascal does not assume any initial value for memory space allocated to a function result unless it is a pointer to a type that occupies more than four bytes of memory.

Register conventions

Registers D0, D1, D2, A0, and A1 are considered scratch registers and are not preserved across procedure calls. All other registers are preserved by the called routine. Register A5 is the global data pointer, register A6 the local frame pointer.

C calling conventions

This section describes the treatment of C parameters, C function results, and C register conventions.

C parameters

Parameters to C functions are evaluated from right to left and are pushed onto the stack in the order they are evaluated. Characters, integers, and enumerated types are passed as sign-extended 32-bit values. Pointers and arrays are passed as 32-bit addresses. Types float, double, comp, and extended are passed as extended 80-bit values (or as 96-bit values for -MC68881). Structures are also passed on the stack. Their size is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The caller removes the parameters from the stack.

C function results

Characters, integers, enumerated types, and pointers are returned in register D0 using as many significant bits as are required by the type of the result. Types float, double, comp, and extended are returned as extended values in registers D0, D1, and A0 (or FPO for -MC68881). The low-order 16 bits of D0 contain the sign and exponent bits, register D1 contains the high-order 32 bits of the significand, and register A0 contains the low-order 32 bits of the significand. Structure values are returned by moving the value into a location, the address of which is passed to the routine as if it were the first parameter (that is, on top of the stack before the JSR). This scheme differs from MPW C 2.0.

C register conventions

Registers `D0`, `D1`, `D2`, `A0`, and `A1` (and `FP0`, `FP1`, `FP2`, and `FP3` for –mc68881) are scratch registers that are not preserved by C functions. All other registers are preserved. Register `A5` is the global frame pointer, register `A6` is the local frame pointer, and register `A7` is the stack pointer. Local stack frames are not necessarily created for simple functions, except when debugger symbol information is being preserved. This scheme differs from MPW C 2.0.

Interfacing C functions to Pascal

Access to routines implemented in C is obtained by declaring the equivalent procedures or functions as "`C; EXTERNAL;`" in the Pascal source text. Such procedures may then be called using a normal Pascal calling sequence (with some caveats shown in Table F-3). The Pascal Compiler arranges the parameters to the C methodology automatically.

Follow the guidelines shown in Table F-3 when using the "`C; EXTERNAL;`" directives.

■ **Table F-3** C-compatible Pascal types

C type	Pascal type
`boolean` (typedef)	`boolean` (*Note:* Only function result values of zero or one will be interpreted correctly by Pascal. Some C functions may return other values.)
`enum`	enumerated (*Note:* If the enumerated type does not have contiguous values, "dummy" values will need to be declared in Pascal.)
`char`	–128..127 subrange (*Note:* If the intention is to denote the extended Macintosh character set, the result may have to be "normalized" by adding +256 to negative values. One would also do this to convert to the Pascal type `char`.)
`unsigned char`	`char`
`short`	`integer`

(Continued)

■ **Table F-3** (Continued) C-compatible Pascal types

C type	Pascal type
`unsigned short`	`longint` (*Note:* This must be a `longint` because of Pascal's range checking. Pascal does not have an unsigned integer type.)
`float`	`real`
`double`	`double`
`comp`	`comp`
`extended`	`extended`
`x*`	`VAR` of `x'` or pointer to `x'`, where `x'` is the Pascal type corresponding to the C type `x`. Also may convert to pointer to array of `x'` when C intends an array to be passed.
`char*`	pointer to `ARRAY OF char`
`int, long`	`longint`
`unsigned int, unsigned long`	`longint` (*Note:* If the C declaration is `unsigned long`, then negative values of C function results will be interpreted incorrectly by the receiver.)
`struct`	`RECORD`
`union`	`variant RECORD`
`array`	`ARRAY` (*Note:* C function results of type `ARRAY` are not allowed.)
`unsigned char`	`SET OF` 0..7 (or byte size)
`unsigned short`	`SET OF` 0..15 (or word size)
`unsigned long`	`SET OF` 0..31 (or long size)

Note: Use of the type STRING in Pascal declarations of C functions is not supported.

Examples of functions declared with the C directive

C function	Pascal declaration

`int scani(t) char t;`	`TYPE signedByte = -128..127;`
	`FUNCTION scani(t: signedByte): longint; C;`
	`EXTERNAL;`

`char expos(d, i)` `double d; int*i;`	`FUNCTION expos(d: double; VAR i: longint):` `signedByte; C; EXTERNAL`

Appendix G **The SANE Library**

THIS APPENDIX DESCRIBES THE STANDARD APPLE NUMERIC ENVIRONMENT (SANE) and the routines contained in the SANE libraries SANELib.o and SANELib881.o. It contains two parts:

- a discussion of the data types provided by SANE
- a description of each of the types, procedures, and functions contained in the SANE libraries

SANE is the basis for floating-point mathematical calculations performed by MPW Pascal. It meets all requirements for extended-precision floating-point arithmetic as prescribed by IEEE Standards 754 and 854. It ensures that all floating-point operations are performed consistently and that they return the most accurate results possible.

SANE provides an easy-to-use, flexible environment for floating-point calculations. It gives you extremely accurate results without extra coding. You can write standard ANS Pascal programs using only the `real` type and be confident that your results are as accurate as possible within that format.

If you are interested in precision beyond that possible using only the `real` type, you can use the other floating-point types provided as an extension to Pascal by SANE. In addition, the SANE library contains numerical functions not found in standard Pascal and routines for controlling the environment in which floating-point calculations are performed.

For complete details about SANE, see the *Apple Numerics Manual.* For information about the Macintosh ROM routines that perform fixed-point calculations, see *Inside Macintosh,* Volume 1, Chapter 16, and Volume 4, Chapter 12.

This appendix discusses the version of SANE that is available on all Macintoshes. On Macintoshes with a 68020 and 68881 (like the Macintosh II), the SANE packages can call the 68881 for basic arithmetic functions. This appendix also provides information on the 68881 transcendental functions.

See "SANE and the 68881" later in this appendix and Chapter 13. ■

The SANE data types

The original specification for Pascal called for only one data type for use with floating-point numbers: the `real` type. MPW Pascal extends the language with four types: `single`, `double`, `extended`, and `comp`.

◆ *Note:* The ANSI Standard specifies that any implementation of Pascal that meets its requirements must include a type named `real`. The MPW Pascal `single` type and the ANS Pascal `real` type are synonymous. The body of this manual refers to the `real` type, while the same type is often referred to as `single` in this appendix. The MPW Pascal Compiler will accept either of these terms and treats them in exactly the same way.

Descriptions of the types

The `single` type is the smallest format for use with floating-point numbers. It stores floating-point numbers using 32 bits of storage.

The `double` type is twice the size of the `single` type. It uses 64 bits for storage.

The `extended` type is larger yet—it uses an 80-bit format. All arithmetic involving real-type values is done using the `extended` type. The `extended` type occupies 96 bitswhen the `-MC68881` option is invoked.

The `comp` type stores integral values in a 64-bit format. It's classified as a real type because extended precision arithmetic is done with operands of type `comp` and uses the `extended` type. Results assigned to a variable of type `comp` are converted from `extended`.

The `single`, `double`, and `extended` types are defined by the IEEE Standard.

Choosing a data type

Typically, picking a data type requires that you determine the trade-offs among

- fixed-point and floating-point form
- precision
- range
- memory usage

The precision, range, and memory usage for each SANE data type are shown in Table G-1.

Many programs require a counting type that counts things (pennies, dollars, widgets) exactly. Using SANE, you can write a program that deals with monetary values by representing these values as integral numbers of cents or mils, which can be stored exactly in the `comp` type. The sum, difference, or product of any two `comp` values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9223372036854775807). In addition, `comp` values (for example, the results of accounting computations) can be mixed with `extended` values in floating-point computations (such as compound interest).

Arithmetic with `comp` type variables, like all SANE arithmetic, is done internally using the `extended` type for arithmetic. There is no loss of precision, as conversion from `comp` to `extended` is always exact. You can save space by storing numbers in the `comp` type, which is shorter than `extended`.

Values represented

The floating-point types (`single`, `double`, and `extended`) store binary representations of a sign (+ or –), an exponent, and a significand. A represented number has the value

$\pm\ significand\ *\ 2^{exponent}$

where for normalized numbers, the significand has a single bit to the left of the binary point (that is, $0 \leq significand < 2$).

Range and precision of SANE types

The range and precision of the real types supported by SANE and MPW Pascal are shown in Table G-1. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

■ **Table G-1** SANE data types

Type identifier	Single	Double	Comp	Extended*
Size (bytes:bits)	4:32	8:64	8:64	10:80*
Binary exponent range				
Minimum	–126	–1022	—	–16383
Maximum	127	1023	—	16383
Significand precision				
Bits	24	53	63	64
Decimal digits	7–8	15–16	18–19	19–20
Decimal range (approximate)				
Min negative	–3.4E+38	–1.7E+308	–9.2E18	–1.1E+4932
Max neg norm	–1.2E–38	–2.3E–308		–1.7E–4932
Max neg denorm		–1.5E–45	–5.0E–324	–1.9E–4951
Min pos denorm		1.5E–45	5.0E–324	1.9E–4951
Min pos norm	1.2E–38	2.3E–308		1.7E–4932
Max positive	3.4E+38	1.7E+308	≈9.2E18	1.1E+4932
Infinities	Yes	Yes	No	Yes
NaNs	Yes	Yes	Yes	Yes

**When `-MC68881` is invoked, the `extended` type occupies 12 bytes, or 96 bits. There are no other changes in the data types in this table. See Appendix N for more information.*

Example

Using the `single` type, the largest representable number has

significand = $2 - 2^{-23}$
= $1.11111111111111111111111_2$
exponent = 127
value = $(2 - 2^{-23}) * 2^{127}$
≈ $3.403 * 10^{38}$

the smallest representable positive normalized number has

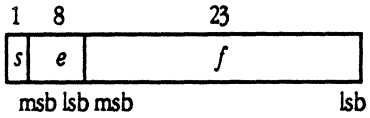
significand = 1
= $1.00000000000000000000000_2$
exponent = -126
value = $1 * 2^{-126}$
≈ $1.175 * 10^{-38}$

and the smallest representable positive denormalized number has

significand $= 2^{-23}$
$= 0.00000000000000000000001_2$
exponent $= -126$
value $= 2^{-23} * 2^{-126}$
$\approx 1.401 * 10^{-45}$

The single type

A 32-bit `single` number is divided into three fields:

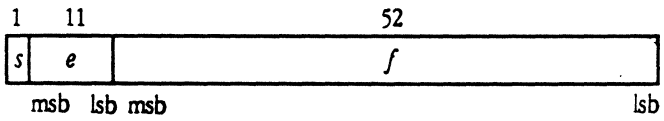


The value v of the number is determined by these fields:

If $0 < e < 255$, then $v = (-1)^s * 2^{(e-127)} * (1.f)$.
If $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-126)} * (0.f)$.
If $e = 0$ and $f = 0$, then $v = (-1)^s * 0$.
If $e = 255$ and $f = 0$, then $v = (-1)^s * \infty$.
If $e = 255$ and $f \neq 0$, then v is a NaN.

The double type

A 64-bit `double` number is divided into three fields:

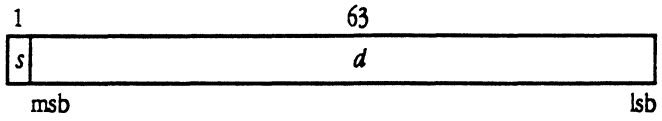


The value v of the number is determined by these fields:

If $0 < e < 2047$, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$.
If $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-1022)} * (0.f)$.
If $e = 0$ and $f = 0$, then $v = (-1)^s * 0$.
If $e = 2047$ and $f = 0$, then $v = (-1)^s * \infty$.
If $e = 2047$ and $f \neq 0$, then v is a NaN.

The comp type

A 64-bit `comp` number is divided into two fields:

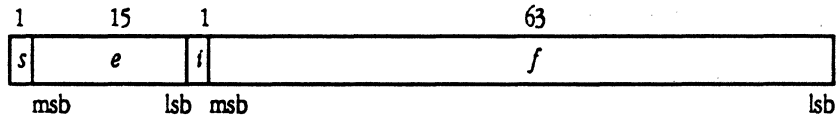


The value *v* of the number is determined by these fields:

If $s = 1$ and $d = 0$,	then *v* is the unique `comp` NaN.
Otherwise,	*v* is the 2's-complement value of the 64-bit representation.

The extended type

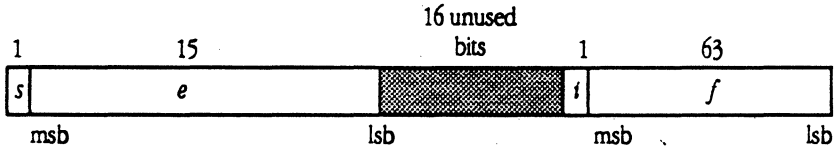
An 80-bit `extended` format number is divided into four fields:



The value *v* of the number is determined by these fields:

If $0 \le e < 32767$,	then $v = (-1)^s = 2^{(e-16383)} * (i.f)$.
If $e = 32767$ and $f = 0$,	then $v = (-1)^s = \infty$, regardless of *i*.
If $e = 32767$ and $f \neq 0$,	then *v* is a NaN, regardless of *i*.

The 96-bit `extended` format used when `-MC68881` is invoked differs from the 80-bit format only in the inclusion of 16 bits of unused space.



Extended arithmetic

While the MPW Pascal types `single`, `double`, and `comp` are intended for economical data storage, the `extended` type is the foundation for all arithmetic computation. As specified by the IEEE Standard, all basic arithmetic operations, including addition, subtraction, multiplication, division, and square root extraction, yield the best possible results. In MPW Pascal, these operations produce `extended` results, so they are accurate to a precision of 19 decimal digits throughout a range exceeding 10^{-4900} to 10^{+4900}.

MPW Pascal takes advantage of `extended` arithmetic by storing all noninteger numeric constants in the `extended` format and by evaluating all noninteger numeric expressions to `extended`, regardless of the types involved. For example, the entire right side of the assignment below will be computed in `extended` before being converted to the type of the left side:

```
VAR
 x, a, b, c: real;
BEGIN
 ---
 x := (b+sqrt(b*b-a*c))/a;
 ---
END.
```

With no special effort by the programmer, MPW Pascal performs computations using the precision and range of the `extended` type. Extra precision means smaller roundoff errors so that results are more accurate, more often. Extra range means overflow and underflow are less common so that programs work more often.

By following a few simple programming practices, you can exploit the `extended` type beyond what MPW Pascal does for you automatically.

Declare variables used for intermediate results to be of type `extended`. This practice is illustrated in the following example, which computes a sum of products:

```
VAR
 Sum: real;
 X, Y: ARRAY[1..n] OF real;
 I: integer;
 T: extended; {to hold intermediate results}
BEGIN
 ---
 T := 0.0;
 FOR I := 1 TO n DO T := T+X[I]*Y[I];
 Sum := T;
 ---
END.
```

Had T been declared `real`, like the input arrays X and Y and the result Sum, each time through the loop the assignment to T would have caused a roundoff error at the limit of `single` precision. In the example, all roundoff errors are at the limit of `extended` precision, except for the one caused by the assignment of T to Sum. This means roundoff errors will be less likely to accumulate to produce an inaccurate result.

Declare formal value parameters and function results to be of type `extended`, rather than `real`, `double`, or `comp`. This saves MPW Pascal from having to do unnecessary conversions between numeric types, which may result in loss of accuracy. The example below illustrates this practice.

```
FUNCTION Area(Radius: extended): extended;
 BEGIN
  Area := Pi*Radius*Radius
 END;
```

Special cases

Although use of the `extended` type makes programs work more often, exceptional cases do arise. Your programs may contain statements like these:

```
Average := Sum/Count;
Area := Side*Side;
```

where all the variables are of type `real`. What happens if `Count` is zero, if `Count` and `Sum` are both zero, or if the product of `Side*Side` is too large to be represented in the `real` format?

MPW Pascal assigns special values to `Average` and `Area`, and your program continues. In fact, the IEEE Standard refers to "exceptions" rather than "errors" and specifies "no halts" as the default mode of operation for its arithmetic. If you need to re-install the IEEE defaults, you would use this statement:

```
SetEnvironment(IEEEDefaultEnv);
```

Note that the `Environment` type is not the `integer` type when the `-MC68881` option is invoked, so the older usage, `SetEnvironment(0)`, will no longer work with this option. For compatibility, `SetEnvironment(IEEEDefaultEnv)` works correctly regardless of the setting of the `-MC68881` option.

The SANE library also contains functions and procedures for determining when exceptional cases occur.

Number classes

Representations in the SANE data formats fall into five classes:

- normalized numbers—like 3.0, 75.8, –2.3e78, and all others that can be represented with a leading significand bit of 1
- zero—+0 and –0
- infinities—positive and negative infinity
- NaNs—short for Not a Number
- denormalized numbers—nonzero numbers that are too small for normalized representation

Infinities

Infinities are special SANE representations that can arise in two ways from operations on finite values:

- When an operation should produce an exact mathematical infinity (such as `1/0`), the result is an infinity.
- When an operation produces a number with magnitude too great for the number's intended floating-point format, the result may (depending on the current rounding direction) be an infinity.

MPW Pascal predefines the constant `inf` to have the value positive infinity. The string constant `inf` also represents infinity for input and output of floating-point values. Infinities behave like mathematical infinities; for example, `1-inf=-inf`. Infinities can be helpful even when "infinity arithmetic" is not the goal; for example, if `x * x` is too large for the `extended` format, the expression `1 + 1/(x * x)` still computes to the correct value of one (assuming overflow halts are off).

Try this:

```
PROGRAM UseInf;
 USES SANE;
 VAR
  X: extended;
 BEGIN
  X := 1e4000;
  Writeln('X*X=',X*X);
  Writeln('1/(X*X)=',1/(X*X))
  Writeln('1+1/(X*X)=',1+1/(X*X))
 END.
```

NaNs

Another special SANE representation is a NaN (Not a Number). A NaN is produced whenever an operation cannot yield a meaningful numeric result. For example, `0/0` and `Sqrt(-1)` produce NaNs.

When a NaN is generated by software SANE or by routines in a SANE library, an associated NaN code is returned as part of the NaN's representation. This code tells you what kind of operation caused the NaN to be created. NaN codes, shown in Table G-2, can help with debugging.

■ **Table G-2** NaN codes

Code	Meaning
1	Invalid square root, such as `Sqrt(-1)`
2	Invalid addition, such as `(+inf) - (+inf)`
4	Invalid division, such as `0/0`
8	Invalid multiplication, such as `0 * inf`
9	Invalid remainder or MOD, such as *x* MOD 0
17	Attempt to convert invalid ASCII string
20	Result of converting the `comp` NaN to floating-point format
21	Attempt to create a NaN with a zero code
33	Invalid argument to trig routine
34	Invalid argument to inverse trig routine
36	Invalid argument to log routine
37	Invalid argument to x^i or x^y routine
38	Invalid argument to financial function

All NaNs generated by the 68881 will have a NaN code of 255.

△ **Important** When the -MC68881 option is set, many of the NaNs in Table G-2 are generated by the 68881; therefore, you will not receive all the settings listed in this table. △

The statement `Writeln(0/0)` will produce the result `NAN(004)` (provided the invalid operation halt is off). `NAN(004)`, `nan(4)`, and `NaN` are examples of acceptable input for reading a NaN into a SANE variable.

Denormalized numbers

Whenever possible, SANE stores values in normalized form: the most significant bit of the significand is a one, rather than a zero.

However, when a very small number is being stored and the exponent is the smallest possible negative value, it is possible to store still smaller values by storing leading zeros. For example,

$1.0..0_2 * 2^{-126}$—smallest normalized real
$0.1..0_2 * 2^{-126}$—still smaller denormalized real

Because of denormalized numbers, IEEE arithmetic has the desirable property that $a <> b$ if and only if $a - b <> 0$. In most non-IEEE arithmetics, $a - b$ will "flush to zero" if $a - b$ is too small for normalized representation, even though a and b may be different values.

Exceptional conditions

Exceptional conditions can arise from floating-point calculations in a number of cases. For example, multiplying two very large values can result in a value too large to be represented in one of the MPW Pascal data formats. Or an operation such as `0/0` can be performed.

SANE provides a way for a program to determine when a floating-point calculation has resulted in one of these exceptional conditions. Exceptional conditions fall into five categories:

- invalid operation
- underflow
- overflow
- divide-by-zero
- inexact

Invalid operation

The invalid operation exception arises when operands for an operation are invalid, so that a meaningful numeric result is impossible. For example, `0/0` and `Sqrt(-1)` are invalid operations.

Underflow

Underflow occurs when a result is both denormalized and has lost significant digits through rounding. For example, to return the result of

$(1.00000000000000000000001_2 * 2^{-126}) / 2$

to the real format, a leading zero would be introduced and the last significant bit would be lost in rounding. The result

$0.10000000000000000000000_2 * 2^{126}$

would be returned and underflow would be signaled.

Overflow

Overflow occurs when a value is calculated that is too large to fit in the format of its designated type. The destination format must be one of the floating-point types; if the destination format is an integer type, the invalid exception occurs.

Divide-by-zero

The divide-by-zero exception occurs when a finite nonzero number is divided by zero. It also occurs when an operation on finite operands produces an exact infinite result. For example, the operation `1/0` (which results in `INF`) and the operation `Ln(0)` (which results in `-INF`) both signal divide-by-zero.

Inexact

The inexact exception occurs when the rounded result of an operation is not identical to the mathematical (exact) result. (Thus any time overflow or underflow occurs, the inexact exception is signaled.) For example, the operation `2/3` signals inexact, regardless of the floating-point format used.

The SANE environment

The SANE "environment" consists of

- rounding direction
- rounding precision
- exception flags
- halt settings

The SANE library includes procedures and functions that allow you to determine the current status of the environment. These procedures and functions can be used to flag exceptional conditions and to control optional environment settings. For example, you may be working with very small values and need to know exactly when underflow occurs. Or you might want to have floating-point conversions rounded downward.

The SANE interfaces and libraries

This section explains each of the constants, types, functions, and procedures contained in the SANE interfaces and libraries. The SANELib881.o library contains the same procedures and functions available in SANELib.o but must be used when you have invoked the `-MC68881` compiler option. Most of the actual SANE code is in `Pack4` and `Pack5`. SANELib.o and SANELib881.o contain code interfaces to `Pack4` and `Pack5`. The file SANE.p contains the interface text for SANE.

Descriptions of constants and types

Each of the constants and types defined by the SANE interface is briefly discussed in the following section. For more information, see the descriptions of the specific procedures and functions mentioned.

The DecStrLen constant

`DecStrLen` (Decimal String Length) is defined by

```
DecStrLen = 255;
```

`DecStrLen` is the constant that defines the maximum length of a decimal numeric string. It is the size attribute of variables of type `DecStr`.

Exception condition constants

The declarations that define the five exception condition constants used when you have not invoked the `-MC68881` option are

```
Invalid = 1;
Underflow = 2;
Overflow = 4;
DivByZero = 8;
Inexact = 16;
```

These constants are used to define the value of a variable of the type `Exception`.

For example, if `e` is a variable of type `Exception`, then

```
e := Invalid + Overflow + DivByZero
```

gives `e` a value that represents these three exceptions collectively.

The `SetException`, `TestException`, `SetHalt`, and `TestHalt` routines all take arguments of type `Exception`.

The declarations that define the 13 exception condition constants used when you have invoked the `-MC68881` option are

```
Inexact = 8;
DivByZero = 16;
Underflow = 32;
Overflow = 64;
Invalid = 128;

CurInex1 = 256;
CurInex2 = 512;
CurDivByZero = 1024;
CurUnderflow = 2048;
CurOverflow = 4096;
CurOpError = 8192;
CurSigNaN = 16384;
CurBSonUnor = 32768;
```

Refer to the *MC68881 Floating-Point Coprocessor User's Manual* for further information about 68881 exceptions.

The DecStr type

The declaration that defines the `DecStr` type is

```
DecStr = STRING[DecStrLen];
```

The `DecStr` type is a string with a size attribute of `DecStrLen`, or 255 characters. It's used to hold the decimal representation, in ASCII characters, of a number.

The DecForm record type

A record of type `DecForm` (Decimal Format) is defined by this declaration:

```
DecForm = RECORD
      Style: (FloatDecimal, FixedDecimal);
      Digits: integer
     END;
```

A `DecForm` record holds the specifications for the format of a decimal number.

- The `Style` field determines whether the decimal representation will be floating-point or fixed-point.
- The `Digits` field holds the number of significant digits for float style or the number of digits to the right of the decimal point for fixed style.

The `Num2Str` procedure takes a `DecForm` argument. It uses the information in `DecForm` to determine the format for the string to be returned.

The RelOp type

The `RelOp` (Relational Operator) type is defined by

```
RelOp = (GreaterThan, LessThan, EqualTo, Unordered);
```

A result of this type is returned by the `Relation` function, described later.

The NumClass type

The `NumClass` type is defined by

```
NumClass = (SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum);
```

■ **Table G-3** Number class descriptions

Number class	Meaning
`SNaN`	Signaling NaN
`QNaN`	Quiet NaN
`Infinite`	Infinity or –Infinity
`ZeroNum`	0 or –0
`NormalNum`	Normalized number
`DenormalNum`	Denormalized number

Quiet NaNs are the usual kind produced by floating-point operations. Signaling NaNs, potentially useful for flagging uninitialized variables, are discussed in the *Apple Numerics Manual.*

The `NumClass` type is used to return results from the inquiry functions, described below.

The Exception type

A variable of type `Exception` holds an `integer` value that corresponds to the value of one of the `Exception` constants, or to a sum of two or more of the `Exception` constants. Unless the `-MC68881` option is set, the `Exception` type is defined by

```
Exception = integer;
```

If the -MC68881 option is set, the Exception type is defined by

```
Exception =longint;
```

The SetException, TestException, SetHalt, and TestHalt routines all take arguments of type Exception.

The RoundDir type

The RoundDir (Rounding Direction) type is defined by

```
RoundDir = (ToNearest, Upward, Downward, TowardZero);
```

The RoundDir type is used to determine how values are to be rounded, when rounding becomes necessary during arithmetic operations or conversions. The SetRound procedure takes an argument of type RoundDir. The GetRound function returns a value of type RoundDir.

The RoundPre type

The RoundPre (Rounding Precision) type is defined by

```
RoundPre = (ExtPrecision, DblPrecision, RealPrecision);
```

Rounding precision can be used to simulate arithmetic with only single or double precision. The SetPrecision procedure takes an argument of type RoundPre. The GetPrecision function returns a value of type RoundPre.

The Environment type

A variable of type Environment holds a value that represents the settings of the SANE environment. For example, a setting of IEEEDefaultEnv represents the default IEEE setting (rounding to the nearest, extended-precision rounding, exceptions clear, and no halts set). Unless the -MC68881 option is set, the Environment type is defined by

```
Environment = integer;
```

You use a variable of type Environment with the environmental access routines SetEnvironment, GetEnvironment, ProcEntry, and ProcExit.

If -MC68881 is set, the Environment type is defined by

```
Environment = RECORD
                FPCR: longint;
                FPSR: longint;
              END;
```

where FPCR stands for the 68881 floating-point control register and FPSR stands for its floating-point status register.

Numeric procedures and functions

This section includes a description of each of the procedures and functions in the SANE libraries. More detailed information can be found in the *Apple Numerics Manual*, which is available from Apple dealers.

Remember that any function with a formal parameter of any of the real types can be passed a value of any real or integer type. Floating-point value parameters in MPW 3.0 Pascal will accept expressions and variables of any of the following types: `integer`, `longint`, `real` (`single`), `double`, `comp`, or `extended`. We abbreviate the list with the term *numeric argument* in the explanatory text below.

Conversions between numeric binary types

The SANE libraries contain functions that convert numeric values (in binary representation) to the binary formats of the `integer`, `longint`, and `extended` types.

The Num2Integer and Num2Longint functions

```
Num2Integer(x: extended): integer;
Num2Longint(x: extended): longint;
```

The `Num2Integer` function takes a numeric argument and returns a result of type `integer`.

The `Num2Longint` function takes a numeric argument and returns a result of type `longint`.

The value returned by these functions depends upon the rounding direction (set using the `SetRound` procedure). Using the standard rounding direction, `ToNearest`, the examples

```
Num2Integer(99.6);
Num2Longint(99.6);
```

return the value 100.0.

`Num2Integer` and `Num2Longint` are similar to the MPW 3.0 Pascal functions `Round` and `Trunc`. However, `Num2Integer` and `Num2Longint` take the current rounding direction into account. The `Round` function always returns the nearest `longint` value; the `Trunc` function always rounds toward zero.

Here's an example of how these functions are used:

```
VAR
 A: extended;
 B, C, D: longint;
BEGIN
 A := 99.999;
 B := Num2Longint(A);
 C := Round(A);
 D := Trunc(A)
END;
```

After this code is executed, both B and C have the value 100 and D has the value 99.

```
BEGIN
 A := 99.999;
 SetRound(Downward);
 B := Num2Longint(A);
 C := Round(A);
 D := Trunc(A)
END;
```

With this code, however, the values of B and D are 99. But the value of C is again 100. The `Round` and `Trunc` functions always calculate their value in the same way, regardless of the rounding direction.

Using the `ToNearest` rounding direction, `Num2Integer` and `Num2Longint` round values halfway between two integers to the nearest even integer (as prescribed by the IEEE Standard). For example, `Num2Integer(2.5)` returns two. The `Round` function rounds these halfway values away from zero. For example, `Round(2.5)` returns three.

The Num2Extended function

```
Num2Extended(x: extended): extended;
```

The `Num2Extended` function can be passed any real-type or integer-type argument. It converts its argument to the `extended` format. This is useful for forcing floating-point arithmetic when all variables involved are of the integer types.

Conversions between decimal strings and binary

The SANE library includes the `Num2Str` procedure and the `Str2Num` function to convert numbers between decimal ASCII character representations and binary.

- *Note:* The MPW 3.0 Pascal input and output procedures, described in Chapter 10, use the routines for Pascal I/O conversions between decimal ASCII and binary.

The Num2Str procedure

```
Num2Str(f: DecForm; x: extended; VAR s: DecStr);
```

The `Num2Str` procedure converts a numeric value *x* to a decimal string, returned in *s*, using the specifications in the `DecForm` record *f*. Here are some examples of how `Num2Str` uses the arguments passed to it to format a string:

■ **Table G-4** Num2Str examples

DecForm.Style	DecForm.Digits	*x*	*s*
`FloatDecimal`	6	`123.45`	`' 1.23450e+2'`
`FloatDecimal`	2	`123.45`	`' 1.2e+2'`
`FixedDecimal`	6	`123.45`	`'123.450000'`
`FixedDecimal`	2	`123.45`	`'123.45'`

The Str2Num function

```
Str2Num(s: DecStr): extended;
```

The `Str2Num` function takes a decimal string argument (of type `DecStr`) and converts it to type `extended`.

Arithmetic, auxiliary, and elementary functions

The SANE library includes a set of functions that supplement the arithmetic functions described in Chapter 12.

The Remainder function

```
Remainder(x, y: extended; VAR quo: integer);
```

The `Remainder` function returns the remainder of the division of its two `numeric` arguments *x*/*y*, as specified by the IEEE Standard. This function returns an exact remainder of the smallest possible magnitude. The result is computed as $x - n*y$, where n is a nearest integral approximation to the quotient x/y. For example, `Remainder(9, 5, q)` returns –1, because $-1 = 9-2*5$.

The `integer` variable argument *quo* receives the seven low-order bits of n as a value between –127 and 127; this is useful for programming functions, like the trigonometric functions, that require argument reduction.

- *Note:* Remember that the Pascal operator MOD can be used only with integral values. The `Remainder` function can be used with either real-type or integer-type values.

The Rint function

```
Rint(x: extended): extended;
```

The `Rint` function takes a numeric argument and rounds it to an integral value in the `extended` format. Note that all sufficiently large floating-point values are integral. The result depends upon the rounding direction, which can be changed by using the `SetRound` procedure.

The Scalb function

```
Scalb(n: integer; x: extended): extended;
```

The `Scalb` function takes two arguments. The first is a value of type `integer`;the second is an `extended` value. The function scales the `extended` value by the power of two specified by the `integer` argument. The value $2^n x$ is returned in `extended` format.

The Logb function

```
Logb(x: extended): extended;
```

The `Logb` function takes a numeric argument and returns the largest power of two that does not exceed its argument's magnitude. For example, `Logb(-65535)` yields 15 because $2^{15} \leq 65535 < 2^{16}$.

The CopySign function

```
CopySign(x, y: extended): extended;
```

The `CopySign` function takes numeric arguments. It returns a value equal to the second argument, but with the sign of the first argument. For example, `CopySign(2.0, -3.0)` yields 3.0. The `CopySign` function returns an `extended` value.

The NextReal function

```
NextReal(x, y: real): extended;
```

The `NextReal` function takes two `real` arguments. It returns the next value that can be represented in the `real` format after the first argument, in the direction of the second argument. It returns an `extended` value.

- *Note:* Although `NextReal`, `NextDouble`, and `NextExtended` can take any numeric argument, `NextReal` internally converts its arguments to `real`, `NextDouble` to `double`, and `NextExtended` to `extended`.

The NextDouble function

```
NextDouble(x, y: double): extended;
```

The `NextDouble` function takes two `double` arguments. It returns the next value that can be represented in the `double` format after the first argument, in the direction of the second argument. It returns an `extended` value.

The NextExtended function

```
NextExtended(x, y: extended): extended;
```

The `NextExtended` function takes two `extended` arguments. It returns the next value that can be represented in the `extended` format after the first argument, in the direction of the second argument. It returns an `extended` value.

The Log2 function

```
Log2(x: extended): extended;
```

The `Log2` function takes a numeric argument and returns the base-2 logarithm of its argument in `extended` format.

The Ln1 function

```
Ln1(x: extended): extended;
```

The `Ln1` function takes a numeric argument and returns the base-e logarithm of one plus the argument: `Ln(1 + x)`. It returns an `extended` value. For *x* near zero, `Ln1(x)` is more accurate than `Ln(1.0 + x)`.

The Exp2 function

```
Exp2(x: extended): extended;
```

The `Exp2` function takes a numeric argument and returns two raised to the power of the argument: 2^x. It returns an `extended` value.

The Exp1 function

```
Exp1(x: extended): extended;
```

The `Exp1` function takes a numeric argument and returns $e^x - 1$. It returns an `extended` value. For *x* near zero, `Exp1(x)` is more accurate than `Exp(x) - 1.0`.

The XpwrI function

```
XpwrI(x: extended; i: integer): extended;
```

The `XpwrI` function takes one `numeric` argument and one `integer` argument. It returns the value of its first argument, raised to the power specified by the `integer` argument: x^i. It returns an `extended` value.

The XpwrY function

```
XpwrY(x, y: extended): extended;
```

The `XpwrY` function takes two `numeric` arguments. It returns the value of the first argument, raised to the power specified by the second argument: x^y. It returns an `extended` value.

Financial functions

SANE provides two functions that can be used in financial applications: the `Compound` function and the `Annuity` function.

The Compound function

```
Compound(r, n: extended): extended;
```

The `Compound` function takes two `numeric` arguments. The first argument specifies the interest rate; the second specifies the number of periods compound. It returns $(1 + r)^n$, which is the principal plus accrued compound interest on an original investment of one unit. It returns an `extended` value.

The Annuity function

```
Annuity(r, n: extended): extended
```

The `Annuity` function takes two `numeric` arguments. The first argument specifies the interest rate; the second specifies the number of periods. `Annuity` returns $(1 - (1 + r)^{-n}) / r$, which is the present value factor of an ordinary annuity. It returns an `extended` value. Here is an example of how the `Annuity` function can be used:

```
PROGRAM Loan;
 VAR
  Loan, Payment, Interest, Periods: extended;
 BEGIN
  writeln('Loan amount: ');
  readln(Loan);
  writeln('Annual interest rate (Enter as a decimal.): ');
  readln(Interest);
  writeln('Number of years: ');
  readln(Periods);
  Payment := Loan/Annuity(Interest/12, Periods*12);
  write('Your payment is. ');
  write(Payment:8:2)
 END.
```

In this example, given a loan amount of $120,000 and an interest rate of 0.1075 for 30 years, the monthly payment will be $1120.18.

Trigonometric functions

MPW 3.0 Pascal includes the predefined `Sin`, `Cos`, and `Arctan` functions. In addition, the SANE library provides the `Tan` function.

The Tan function

```
Tan(x: extended): extended;
```

The `Tan` function returns the tangent of a numeric argument. Note that the argument must be expressed in radians. The `Tan` function returns an `extended` value.

Additional transcendental routines

MPW 3.0 Pascal predefines these nine routines when the `-MC68881` Compiler option is used. These routines are not part of the SANE interface and are currently unavailable without the `-MC68881` option.

The Arctanh function

```
Arctanh (x: extended): extended;
```

The `Arctanh` function returns the hyperbolic arctangent of a numeric argument. The `Arctanh` function returns an `extended` value.

The Cosh function

```
Cosh(x: extended): extended;
```

The `Cosh` function returns the hyperbolic cosine of a numeric argument. The `Cosh` function returns an `extended` value.

The Sinh function

```
Sinh(x: extended): extended;
```

The `Sinh` function returns the hyperbolic sine of a numeric argument. The `Sinh` function returns an `extended` value.

The Tanh function

```
Tanh(x: extended): extended;
```

The `Tanh` function returns the hyperbolic tangent of a numeric argument. The `Tanh` function returns an `extended` value.

The Log10 function

```
Log10(x: extended): extended;
```

The `Log10` function returns the base-10 logarithm of a numeric argument. The `Log10` function returns an `extended` value.

The Exp10 function

```
Exp10(x: extended): extended;
```

The `Exp10` function returns ten raised to the power of a numeric argument: 10^x. The `Exp10` function returns an `extended` value.

The Arccos function

```
Arccos(x: extended): extended;
```

The `Arccos` function returns the principal value, in radians, of the arccosine of a numeric argument. The `Arccos` function returns an `extended` value.

The Arcsin function

```
Arcsin(x: extended): extended;
```

The `Arcsin` function returns the principal value, in radians, of the arcsine of a numeric argument. The `Arcsin` function returns an `extended` value.

The SinCos procedure

```
SinCos(VAR s,c: extended; x: extended);
```

The `SinCos` procedure simultaneously sets the variable *s* to `Sin(x)` and the variable *c* to `Cos(x)`. The parameter *x* is a numeric argument. The `SinCos` procedure is faster than separate function calls to `Sin` and `Cos`.

Inquiry functions

SANE includes several functions that allow you to determine the class of a numeric value. The result of each of these functions is of type `NumClass`, described above. In addition, they include a function that returns the sign of a numeric argument.

The ClassReal function

```
ClassReal(x: real): NumClass;
```

The `ClassReal` function determines the number class of a numeric argument as if the argument were converted to `real` format. For example,

```
ClassReal(1)
ClassReal(1e-310)
```

The first function call returns `NormalNum`, the code for a normalized number. The second call returns `ZeroNum`, the code for zero (because `1e-310` rounds to +0 in the `real` format).

The ClassDouble function

```
ClassDouble(x: double): NumClass;
```

The `ClassDouble` function determines the number class of a numeric argument as if the argument were converted to a `double` format. The result is of type `NumClass`. For example,

```
ClassDouble(0.0/0.0)
ClassDouble(1e-310)
```

The first example returns `QNaN` , the code for a quiet NaN. The second example returns `DenormalNum` , the code for a denormalized number (because `1e-310` is denormalized in the `double` format).

The ClassExtended function

```
ClassExtended(x: extended): NumClass;
```

The `ClassExtended` function determines the number class of a numeric argument as if the argument were converted to an extended format. The result is of type `NumClass`. For example,

```
ClassExtended(1/0)
ClassExtended(e-310)
```

The first example returns `Infinite` , the code for infinities. The second example returns `NormalNum` , the code for a normalized number.

The ClassComp function

```
ClassComp(x: comp): NumClass;
```

The `ClassComp` function determines the number class of its `numeric` argument as if the argument were converted to `comp` format. The result is of type `NumClass`. For example,

```
ClassComp(1)
ClassComp(0.1)
```

The first example returns `NormalNum`, the code for a normal number. The second example returns `ZeroNum`, the code for zero. (Remember that `comp` stores integral values.)

The SignNum function

```
SignNum(x: extended): integer;
```

The `SignNum` function takes a numeric argument and returns an `integer` value that indicates the sign of the argument. The value returned is one if the argument's sign is negative, zero if the argument's sign is positive.

The RandomX function

```
RandomX(VAR x: extended): extended;
```

The `RandomX` function takes a variable parameter of type `extended` that must contain an integral value in the range $1 \leq x \leq 2^{31} - 2$. It returns the next random number (in `extended` format) in sequence within the same range. The variable argument is updated to the value returned. `RandomX` uses this algorithm:

```
NewX = (7^5 * OldX) MOD (2^31-1)
```

The NaN function

```
Nan(x: integer): extended;
```

The `Nan` function takes an `integer` argument and returns a NaN, in `extended` format, associated with the code given as an argument. The SANE NaN error codes are shown in Table G-2 in the section "NaNs," earlier in this chapter.

The Relation function

```
Relation(x, y: extended): RelOp;
```

The `Relation` function takes two `numeric` arguments and returns a value of type `RelOp`. The value returned specifies the relationship between the two arguments.

For example, `Relation(0.1, NaN(0))` returns `Unordered`, as all comparisons involving NaNs are unordered. `Relation(1,3.9)` returns `LessThan`.

Environmental access procedures and functions

The SANE environmental access routines allow you to determine how calculations are to be performed and how to respond to exceptional conditions. The environment consists of

- the rounding direction
- the rounding precision
- exception flags
- halt settings

The rounding direction

The rounding direction can be set in four ways:

- `ToNearest`
- `Upward`
- `Downward`
- `TowardZero`

The default rounding direction is `ToNearest` . You can find out what the current rounding direction is by using the `GetRound` function. You can change the rounding direction by using the `SetRound` procedure.

The GetRound function

```
GetRound: RoundDir;
```

The `GetRound` function returns the current rounding direction as a value of type `RoundDir` .

The SetRound procedure

```
SetRound(r: RoundDir);
```

The `SetRound` procedure takes an argument of type `RoundDir`. The procedure sets the effective rounding direction to the one indicated by the argument.

For example, the code below saves the current rounding direction, computes a function using `TowardZero` rounding, and finally restores the saved rounding direction.

```
VAR
 R: RoundDir;
 X, Y: extended;
BEGIN
 ---
 R := GetRound;
 SetRound(TowardZero);
 Y := f(X);
 SetRound(R);
 ---
END.
```

Rounding precision

You may find that you want to use SANE for performing calculations and simulating the results you would get if you used a system that did not provide extended-precision arithmetic. Normally, all MPW 3.0 Pascal floating-point calculations return results that are rounded to extended precision and range. However, the rounding precision can be set to single or double precision and range. Results will still be returned in the `extended` format. There is no performance benefit in setting single or double rounding precision. You can access the rounding precision by using the `SetPrecision` procedure and the `GetPrecision` function.

The GetPrecision function

```
GetPrecision: RoundPre;
```

The `GetPrecision` function returns a value of type `RoundPre`, which indicates the current rounding precision.

The SetPrecision procedure

```
SetPrecision(p: RoundPre);
```

The `SetPrecision` procedure takes an argument that is a variable of type `RoundPre`, which indicates the desired rounding precision.

Exceptions

When `-MC68881` is not set, exceptional results arising from numeric calculations fall into five categories. The file SANE.p in PInterfaces defines a constant for each kind of exception, as shown in Table G-5.

■ **Table G-5** SANE exceptions

Exception	Constant value	When arising	Example
`Invalid`	1	Operation not meaningful—NaN result	Sqrt(-1)
`Underflow`	2	Accuracy lost—result too small	$2^{-16383}/3$
`Overflow`	4	Result too large for number representation	2^{16384}
`DivByZero`	8	Division of nonzero number by zero	1/0
`Inexact`	16	Rounded result not same as exact math result	1/3

The exceptions shown in Table G-5 occur under the following conditions:

- If an invalid operation is performed, `Invalid` is set.
- If underflow occurs, `Underflow` is set.
- If overflow occurs, `Overflow` is set.
- If division by zero occurs, `DivByZero` is set.
- If the result of the calculation is inexact, `Inexact` is set.

Table G-6 lists the exceptional results that can occur when `-MC68881` is set. The prefix `Cur` stands for current exception. These eight values match the 68881 Exception Status byte. See the *Motorola MC68881 Floating-Point Coprocessor User's Manual* for details.

■ **Table G-6** 68881 SANE exceptions

Exception	Constant value
`Inexact`	8
`DivByZero`	16
`Underflow`	32
`Overflow`	64
`Invalid`	128
`CurInex1`	256
`CurInex2`	512
`CurDivByZero`	1024
`CurUnderflow`	2048
`CurOverflow`	4096
`CurOpError`	8192
`CurSigNaN`	16384
`CurBSonUnor`	32768

The SetException procedure

```
SetException(e: Exception; b: boolean);
```

The `SetException` procedure takes one argument of type `Exception` and a second argument of type `boolean`. If the second argument is `true`, the procedure signals the exceptions encoded in its first argument. If the second argument is `false`, it clears the exception flags specified by the first argument. For example,

```
SetException(Overflow + Inexact, true);
```

signals the `Overflow` and `Inexact` exceptions. If halt on `Overflow` or `Inexact` were set, this statement would halt the program.

The TestException function

```
TestException(e: Exception): boolean;
```

The `TestException` function takes an argument of type `Exception` and returns a `boolean` value that indicates whether any of the exceptions encoded in its argument are set or not.

Following the `SetException` statement above, the statement

```
TestException(OverFlow + Invalid);
```

would return `true`.

Using exceptional conditions to halt a program

The SANE environment includes a halt setting for each of the exceptions that determines whether the occurrence of the exception halts the program. By default, MPW 3.0 Pascal adheres to the IEEE Standard by initializing all halts clear (off).

You can access the halt settings by using the `TestHalt` function and the `SetHalt` procedure.

The TestHalt function

```
TestHalt(e: Exception): boolean;
```

The `TestHalt` function takes an argument of type `Exception` and returns a `boolean` value. If any of the halts indicated by the `Exception` argument are set, the function returns `true`; otherwise it returns `false`.

The SetHalt procedure

```
SetHalt(e: Exception; b: boolean);
```

The `SetHalt` procedure takes two arguments. The first is of type `Exception`. This indicates which exceptions you want to halt your program. The second argument is of type `boolean`. If the value of the `boolean` argument is `true` , occurrences of the indicated exceptions will cause your program to halt. If it is `false` , your program will continue to run when these exceptions occur.

Halts and the 68881

When the `-MC68881` option is set, floating-point halts are generated by the 68881 and not by `Pack4`. Old sources for a halt handler will no longer work when compiled with `-MC68881` set.

To write a halt handler for the 68881, consult the Motorola *MC68881 Floating-Point Coprocessor User's Manual.*

For details on SANE halt handlers compatible with `Pack4`, see the *Apple Numerics Manual*. The following support for 68881 halts and traps has been added to the file SANE.p:

```
TrapVector =          RECORD
     Unordered:       longint;
     Inexact:         longint;
     DivByZero:       longint;
     Underflow:       longint;
     Operator:        longint;
     Overflow:        longint;
     SigNaN:          longint
          END;

PROCEDURE GetTrapVector (VAR Traps: TrapVector);
                  { Traps <-- FPCP trap vectors }

PROCEDURE SetTrapVector (Traps: TrapVector);
                  { FPCP trap vectors <-- Traps }
```

The `TrapVector` type definition and its accompanying procedures, `GetTrapVector` and `SetTrapVector`, give you access to those 68020 exception vectors that manage 68881 floating-point exceptions. For details on 68881 exception vectors, see Motorola's 68020 and 68881 manuals. The procedures for setting and getting halts, exceptions, and environments work as they do when `-MC68881` is not set; they differ only in the data types used as their arguments.

Finally, unlike the `Pack4` halt mechanism, the 68881 supports the IEEE-recommended trapping mechanism. The word *trap* is prominent in the names above to emphasize this difference. IEEE traps sometimes rebias the exponent of floating-point results; in some cases, exceptions are set differently when traps are enabled, and the stack passed to a trap handler is set up differently by the 68881 trap mechanism.

Saving and restoring environmental settings

The entire SANE environment (rounding direction, rounding precision, exception flags, and halt settings) can be encoded in a value of type `Environment`. The procedures described below access the current SANE environment as a whole. They are useful for managing the environment so that routines run with the environments they require and for controlling the exception information passed between routines.

The GetEnvironment procedure

```
GetEnvironment(VAR e: Environment);
```

The `GetEnvironment` procedure takes an argument of type `Environment` as a variable parameter and assigns the current settings of the environment to that variable.

When your program begins, the environment will reflect the MPW 3.0 Pascal defaults:

- rounding direction `ToNearest`
- rounding precision `extended`
- all exception flags cleared
- no halts set

The SetEnvironment procedure

```
SetEnvironment(e: Environment);
```

The `SetEnvironment` procedure takes an argument of type `Environment` . It sets the floating-point environment to the one encoded in its argument. To re-install the default environment, use the statement

```
SetEnvironment(IEEEDefaultEnv);
```

The following procedure ensures that it will run under the IEEE default environment, while not affecting its caller's environment:

```
PROCEDURE P;
 VAR
  SaveEnv: Environment;
 BEGIN
  GetEnvironment(SaveEnv);
  SetEnvironment(IEEEDefaultEnv);
  ---
  SetEnvironment(SaveEnv)
 END;
```

Note that the `Environment` type is not the `integer` type when the `-MC68881` Compiler option is invoked, so the older usage, `SetEnvironment(0)` , will no longer work with this option. For compatibility, `SetEnvironment(IEEEDefaultEnv)` works correctly regardless of the setting of the `-MC68881` option.

The ProcEntry procedure

```
ProcEntry(VAR e: Environment);
```

The `ProcEntry` procedure saves the current environment (the rounding direction, rounding precision, exception flags, and halt settings) in the `Environment` variable passed to the procedure, and then sets the environment to the IEEE defaults. The statement `ProcEntry(e)` is equivalent to

```
GetEnvironment(e);
SetEnvironment(IEEEDefaultEnv);
```

The ProcExit procedure

```
ProcExit(e: Environment);
```

The `ProcExit` procedure takes an argument of type `Environment`. It temporarily saves the current exception flags, sets the effective environment to be the one encoded in its argument, and then signals the temporarily saved exceptions.

`ProcEntry` and `ProcExit` can be used in routines to selectively hide spurious exceptions from the routine's caller. For example,

```
FUNCTION Arccos(x: extended): extended;
 VAR
  e: Environment;
 BEGIN
  ProcEntry(e);
  Arccos := 2.0*Arctan(Sqrt(1.0-x)/(1.0+x));
  SetException(DivByZero, false);
  ProcExit(e)
 END;
```

`ProcEntry(e)` saves the caller's environment in *e* and sets IEEE defaults so that exceptions cannot halt the routine. If `x` = –1, the computation of the right side of the assignment to `ArcCos` will signal `DivByZero`, even though `ArcCos` will be assigned the correct value, `pi`. `SetException(DivByZero, false)` clears the `DivByZero` flag, so the caller never sees it. If `x` > 1 or `x` < –1, the computation of `ArcCos` will appropriately signal `Invalid`. The `ProcExit` procedure will resignal `Invalid` after restoring the caller's environment, so if the caller's environment calls for halts on invalid, the halt will occur.

Support for the 68881

The following list summarizes the MPW 3.0 Pascal support for the 68881 floating-point coprocessor. Detailed information about SANE and programming the 68881 is provided in the sections below.

- Applications compiled without the `-MC68881` option will run on all Macintoshes. If `-MC68881` is set, then your application will run only with a 68020 and a 68881 in the computer.
- One library, SANELib881.o, has been added and the interface file SANE.p has been updated to provide support for the floating-point coprocessor.
- The command-line option `-MC68881` may be used to access the 68881 directly for addition, subtraction, multiplication, division, square root, remainder, binary-binary conversion, and comparison. When this option is set, you must link with SANELib881.o instead of SANELib.o.

- For the same effect, you may type `{$MC68881+}` in the source code (it must appear at the start of the file before any `VAR`, `PROCEDURE`, or `FUNCTION` declarations or any `USES` statements).
- With the `-MC68881` option, the `extended` type becomes 12 bytes long and variables of `extended` type may be allocated to registers.
- By default, the Compiler calls `Pack5` to perform accurate and fully compatible transcendental (elementary) functions even if you have set the `-MC68881` option.
- If you want the Compiler to emit direct calls to the 68881 for less accurate but fully compatible, but very fast transcendental functions, then add `-d Elems881=true` to the Pascal command line.

SANE and the 68881

SANE is Apple's numeric environment. It is a superset of IEEE Standard 754 numerics. All Apple computers and most Apple programming languages incorporate SANE. In the Macintosh family, SANE is contained in the system ROM as `Pack4` and `Pack5` . MPW 3.0 Pascal handles floating-point arithmetic by emitting code to call the packs. Floating-point across the Macintosh family is identical in accuracy and, except for the Macintosh II, about equally fast.

Floating-point performance on the Macintosh II is about an order of magnitude faster than on the Macintosh Plus because `Pack4` and `Pack5` (in the Macintosh II ROM) take advantage of the 68881 floating-point coprocessor whenever feasible. The 68881 is a very fast floating-point chip that conforms fully to IEEE 754 and, like SANE, has several built-in extensions to the IEEE Standard. For the fastest possible floating-point performance, MPW 3.0 Pascal can use the 68881 directly, avoiding calls to `Pack4` and `Pack5`.

The `-MC68881` command line option directs the Compiler to use the 68881 calls for basic arithmetic operations (addition, subtraction, multiplication, square root, remainder, comparison, and binary-binary format conversions). When you invoke this option, the following three effects occur:

- The size of the `extended` data type changes.
- Arithmetic executes about two orders of magnitude faster than on the Macintosh Plus (about one order of magnitude faster than the Macintosh II without the `-MC68881` option).
- Your application will contain 68881 instructions and therefore will no longer run on a Macintosh without a 68881.

Result values will be identical to those received without the `-MC68881` option. A further option, written `-d Elems881=true`, directs the Compiler to use the 68881 calls for all the 68881 transcendental functions (logarithms, exponentials, trigonometric, and hyperbolic trigonometric). This option has no meaning unless the `-MC68881` option is already invoked; in addition, it has two important effects:

- Results will differ from results without the `-d Elems881=true` option because `Pack5` in ROM is more accurate in its transcendental functions than the 68881.
- Transcendental function performance will improve by more than another order of magnitude.

More about the 68881

Pascal allows you to redefine any predefined functions, procedures, and types. Using `-d Elems881=true`, described above, simply controls the redefinition of several routines built into the Compiler. SANE.p in the PInterfaces folder contains declarations of 19 transcendental functions that the Compiler predefines. The transcendental functions fall into three groups:

- *Pascal predefined functions*, including `Sin`, `Cos`, `Arctan`, `Exp`, and `Ln`
- *SANE functions*, including `Tan`, `Expl`, `Exp2`, `Lnl`, and `Log2`
- Additional functions, including `Arctanh`, `Cosh`, `Sinh`, `Tanh`, `Log10`, `Exp10`, `Arccos`, `Arcsin`, and `Sincos`

When `-d Elems881=true` is set (and `-MC68881` is invoked), the Compiler makes direct 68881 calls for all Pascal predefined functions, SANE functions, and additional functions listed above. Otherwise, the declarations for the Pascal predefined functions and the SANE functions are seen by the Compiler in SANE.p, direct calls for these functions are not made, and the Compiler generates calls to the versions of these routines found in SANELib.881.o. In effect, SANE.p tells the Compiler to call the version of these functions that is supplied at link time in the file SANELib881.o in the {Plibraries} folder. SANELib881.o, in turn, calls `Pack5` in the ROM for accuracy and compatibility. (When the `-MC68881` option is set, link only with SANELib881.o; otherwise, link with SANELib.o.)

Register usage

The 68881 adds eight new registers, called FP0,...,FP7, to the familiar D0,...,D7 and A0,...,A7 of the 68020. The FP registers are 96 bits wide and are designed to hold extended-precision data. Pascal optimizes the use of these registers for variables and for expression evaluation. Pascal puts variables in FP4,...,FP7 and allocates FP0,...,FP3 as scratch registers.

For example, all floating-point expressions, such as the right-hand side of

```
x:= 3*y+6;
```

are evaluated in FP registers. If `x` and `y` are variables of the `extended` type, then the Compiler may also allocate them to registers, improving performance even further.

Converting between extended formats in mixed-world programs

Apple recommends compiling all of your program files either with the `-mc68881` option never set or with it always set. Building a program with mixed settings of this option causes problems in the following ways:

- `Extended` constants and variables in some parts of a mixed-setting program are 80 bits long; in other parts, they are 96 bits long. This can cause problems when a portion of your program refers to constants and variables from another portion with the other `extended` format.
- Since floating-point value parameters are always promoted to extended, calling a function that has the other `extended` format can cause trouble.
- The two SANE libraries SANELib.o and SANELib881.o contain routines with identical names. To refer to routines from both libraries in the same link step:
  - Use Lib to build your own library (selecting the routines you need from one of the libraries).
  - Rename the routines and the modules containing them to avoid name collision. This is further complicated by requiring the use of DumpObj to determine the names of the modules containing the entry points whose names need changing.
  - Link with your new library and with the unchanged SANE library.

If you must mix files that contain different extended formats, limited support is offered to solve the problem with calling a function that has the other `extended` format. SANE.p defines the following functions to convert between 96-bit and 80-bit `extended` formats:

```
FUNCTION X96to80(x: extended96): extended;
FUNCTION X80to96(x: extended): extended96;
```

for use in the 80-bit world, and

```
FUNCTION X96to80(x: extended): extended80;
FUNCTION X80to96(x: extended80): extended96;
```

for use in the 96-bit world.

The types

```
Extended80 = ARRAY [0..4] OF integer;
```

and

```
Extended96 = ARRAY [0..5] OF integer;
```

are useful only in these transfer routines—you cannot do arithmetic with them. They are not understood by the Compiler to be equivalent to the `extended` type.

For example, if

```
FUNCTION foo(x: extended): extended;
```

is compiled with the `-MC68881` option set (so that, for this function, the `extended` type is 96 bits wide), then to call `foo` from an 80-bit world, you would

1. Declare foo in the 80-bit interface as follows:
   ```
   function foo(x: extended96): extended96;
   ```
2. Call foo as
   ```
   X96toX80(foo(X80toX96(X)));
   ```

Appendix H **The PasMat Utility**

THIS APPENDIX DESCRIBES PASMAT, an MPW Shell utility program that you can use to convert your source text into "pretty-printed" format.

Syntax

`PasMat` [*option...*] [*inputfile* [*outputfile*]]

Description

Reformats Pascal source code into a standard format, suitable for printouts or compilation. PasMat accepts full programs, external procedures, blocks, and groups of statements.

◆ *Note:* A syntactically incorrect program causes PasMat to abort. If this happens, the generated output will contain the formatted source up to the point of the error.

PasMat options let you do the following:

- convert a program to uniform case conventions
- indent a program to show its logical structure, and adjust lines to fit into a specified line length
- change the comment delimiters `(* *)` to `{ }`
- remove the underscore character `(_)` from identifiers, rename identifiers, or change their case
- format include files named in MPW Pascal include directives

PasMat specifications can be made through its options or through special formatter directives, which resemble Pascal Compiler directives and are inserted into the source file as Pascal comments.

PasMat's default formatting is straightforward and does not necessarily require you to use any options. The best way to find out how PasMat formats something is to try out a small example and see. ■

Input If no input files are specified, standard input is formatted.

Output If no output file is specified, the formatted output is written to the standard output file. Refer to "Error Handling" below for more information about PasMat's treatment of errors in the source.

Options Most of the following options modify the initial default settings of the directives described in the *Macintosh Programmer's Workshop 3.0 Reference* manual.

`-a`	Set `a-` to disable `CASE` label bunching.
`-b`	Set `b+` to enable `IF` bunching.
`-body`	Set `body+` to align procedure bodies with their enclosing `BEGIN...END` pair.
`-c`	Set `c+` for placement of `BEGIN` on same line as previous word.
`-d`	Set `d+` to enable the replacement of `(**)` with `{ }` comment delimiters.
`-e`	Set `e+` to capitalize identifiers.
`-entab`	Replace runs of blanks with tabs. The tab stop value is determined by the `-t` option or current `t=`*n* directive (*not* by the file's tab setting).
`-f`	Set `f-` to disable formatting.
`-g`	Set `g+` to group assignment and call statements.
`-h`	Set `h-` to disable `FOR`, `WHILE`, and `WITH` bunching.
`-i`*pathname* [`,` *pathname...*]	Search for included files in the specified directories. Multiple `-i` options may be specified. At most 15 directories will be searched. The search order is specified under the description of the Pascal command. (Note that `USES` declarations are not processed by PasMat.)

`-in` Set `in+` to process Pascal Compiler includes. This option is implied if the `-i` option is used.

`-k` Set `k+` to indent statements between `BEGIN...END` pairs.

`-l` Set `l+` for literal copy of reserved words and identifiers.

`-list` *filename* Generate a listing of the formatted source. The listing is written to the specified file.

`-n` Set `n+` to group formal parameters.

`-o` *width* Set the output line width. The maximum value allowed is 150. The default is 80.

`-p` Display version information and progress information on the diagnostic file.

`-pattern` *=pattern=replacement=*

Process includes (`-in`) and generates a set of output files with the same include structure as the input, but with new names as specified in the *pattern* and *replacement* strings. The output filenames and Pascal Compiler include directives are generated by editing the input (included) filenames according to the *pattern* and *replacement* strings. *Pattern* is a pathname that is to be looked for in the input file and in each included file (the *entire* pathname is used and case is ignored). If the pattern is found, that sequence of characters is replaced by the *replacement* string. The result is a new pathname, which becomes the name for an output file. For example,

```
PasMat -pattern =OldFile=NewFile=
```

replaces each name containing the string `OldFile` with the string

```
NewFile
```

Any character not contained in the *pattern* or *replacement* strings can be used in place of an equal sign. Note that special characters must be quoted. See the example at the end of this appendix.

`-q` — Set `q+` not to treat `ELSE...IF` sequence specially.

`-r` — Set `r+` to uppercase reserved words.

`-rec` — Indent a record's field list under the identifier the record definition is defining.

`-s` *renameFile* — Rename identifiers. *RenameFile* must be a file containing a list of identifiers and their new names. Each line in this file contains two identifiers of up to 63 characters each: the first is the identifier to be renamed, and the second is the name that will replace all occurrences of the first identifier when creating the output. There must be at least one space or tab between the two identifiers. Leading and trailing spaces and tabs are optional. The case of the first identifier doesn't matter, but the second identifier must be specified exactly as it is to appear in the output. The case of all identifiers not specified in the *renameFile* is subject to the other case options (`-e`, `-l`, `-u`, and `-w`) or their corresponding directives. Reserved words cannot be renamed.

`-t` *tab* — Set the tab amount for each indentation level. If the `-entab` option is also specified, tab characters will actually be generated. The default tab value is two.

`-u`	Rename all identifiers based on their first occurrence in the source. Specifications in the rename (`-s`) file always have precedence over this option—that is, the identifier's translation is based on the rename file rather than on the first occurrence.
`-v`	Set `v+` to put `THEN` on a separate line.
`-w`	Set `w+` to write identifiers in uppercase.
`-x`	Set `x+` to suppress space around operators.
`-y`	Set `y+` to suppress space around `:=`.
`-z`	Set `z+` to suppress space after commas.
`-:`	Set `:+` to align colons in `VAR` declarations (only if a `j` PasMat directive in the source specifies a *width*).
`-@`	Set `@+` to force multiple `CASE` tags onto separate lines.
`"-#"`	Set `#+` for "smart" grouping of assignment and call statements (grouped assignment and call statements on an input line will appear grouped on output). ◆ *Note:* Because `#` is the Shell's comment character, this option must be quoted on the command line.
`-_`	Set `_+` for "portability" mode (underscores are deleted from identifiers).

All options except for `-list`, `-pattern`, `-s`, and `-entab` have directive counterparts. It's recommended that you specify the options as directives in the input source so that you won't have to specify them each time you call PasMat.

{PasMatOpts}

variable:

You can also specify a set of default options in the exported Shell variable `{PasMatOpts}`—PasMat processes these options before it processes the command-line options. `{PasMatOpts}` should contain a string (maximum length 255) specifying the options exactly as you would specify them on the command line (*except* for command-line quoting, which should be omitted; also note that the options `-pattern`, `-list`, `-s`, and `-i`, which require a string parameter, can only be specified on the command line). For example, you can define `{PasMatOpts}` to the Shell (perhaps in the UserStartup file) as follows:

```
Set PasMatOpts "-n -u -r -d -entab -# -o 82 -t 2"
Export PasMatOpts
```

The entire definition string must be quoted to preserve the spaces.

As an alternative to specifying the options directly, you can indicate that the options are stored in a file, by specifying the file's full pathname prefixed with the character `^`:

```
Set PasMatOpts "^pathname"
Export PasMatOpts
```

PasMat will now look for the default options in the specified file. The lines in this file can contain any sequence of command-line options (*except* for `-pattern`, `-list`, `-s`, and `-i`), grouped together on the same or separate lines. The lines may be commented by placing the comment in braces (`{...}`). A typical options file might appear as follows:

```
-n          {group formal params on same line}
-u          {auto translation of id's based on 1st
            occurrence}
-r          {uppercase reserved words}
-d          {leave comment braces alone}
-entab      {place real tabs in the output}
-#          {smart grouping}
-o 82       {output line width}
-t 2        {indent tab value}
```

(Except for the tab value, this example shows the recommended set of options.)

If PasMat does find a default set of options, those options will be echoed as part of the status information given with the `-p` option.

Directives

Directives are specified by special comments included in the Pascal source code. These comments have the form

{ [*directives*] *optional text*}

The directives themselves are either switches with the format

<*character*(*s*)>+

or

<*character*(*s*)>-

or numeric directives with the format

<*character*(*s*)>=<*number*>

For the `j` directive only, the numeric directive can also have the special format

`j`=<*number*>*c*/<*number*>*cc*/<*number*>*c*

where the *c*'s are characters and either or both of the first two entries can be omitted (but not the slashes separating them—for example, //<*number*>*c*).

Multiple directives are separated by commas. Spaces within a directive are not allowed. For example,

```
{[b+,o=95,t=4,r-]}
```

sets the switch `b` on and `r` off and sets the numeric directive `o` to 95 and `t` to 4. Case is ignored in directives.

The following directives are recognized:

`a` Place a statement following a `CASE` label. It will be put on the same line if it fits. The default value is `a+`.

`b` Place a statement following `THEN` or `ELSE`. It will be put on the same line if it fits. The default value is `b-`.

`body` Do not indent a procedure body between `BEGIN...END`. The directive `body+` aligns the procedure body with `BEGIN` and `END`. The default value is `body-`.

`c` Place `BEGIN` on same line as its introductory keyword. If `c+` is specified, then `k-` (the default) should be used. The default value is `c-`.

`d` Replace the comment delimiters `(**)` by `{ }`. The default value is `d-`.

`e` Capitalize first (or only) letter of identifiers and the first letter following a break character (underscore). Retain the break character. This directive overrides the `l` and `w` directives. See also the _ (portability) option below. The default value is `e-`.

`f` Turn formatting on or off. This directive goes into effect immediately following the comment in which it is placed. This is useful for saving hand-formatted portions of a program. The default value is `f+`.

`g` Group statements (*i* per line). This directive is specified either as a switch (`g+` or `g-`) or as a numeric directive (`g=`*i*). For `g=`*i*, the space from the current indentation level to the end of the line is divided into *i* fields, and successive statements are put on the boundaries of successive fields. A statement may take more than one field, in which case the next statement again goes on the boundary of the next field. This is similar to the use of tabs on a typewriter. Any statement that requires more than one line may produce strange results on following statements. The `g=`*i* form affects constant declarations and statements. By specifying the `g+` form, only assignment and call statements are grouped together (if they fit on a line). The `g+` directive has effect only if `g=1` is set. The default values are `g-` and `g=1`.

`h` Bunch a single statement on the same line as `FOR`, `WHILE`, and `WITH` if it fits. Otherwise, indent it on the next line. This also applies to `IF` (without `ELSE`) with the `b-` directive. The default value is `h+`.

`in` Process `INCLUDE {$I` *filename*`}` Pascal Compiler (not PasMat) directives. PasMat provides three ways to process include files, with the third way recommended:

- Process all the include files in the input to produce a single output file. To do this, use the `in+` PasMat directive (or option). As each include Pascal Compiler directive is encountered, it will be output on the line before the output of the included source. However, to avoid reprocessing of this directive by the Pascal Compiler (assuming the output is to be eventually compiled), the *i* in the directive is not output.

- Treat each include file separately. Each file is given individually to PasMat to format. By placing an `in=`*n* PasMat numeric directive at the start of each source input file, you can specify the initial indenting level for the file. Indenting for `in=`*n* will start at column *n*t*, that is, the specified level times the indenting tab value (see the `t` directive). Note that because individual include files need not represent syntactically complete Pascal constructs (for example, an include file can contain a procedure with many nested inner procedures but without the body of the outer procedure), PasMat may report a syntax error. If this happens, check the output to see if the entire include file was processed.

- Process the entire source as in the first method above, but instead of generating a single source with the include directives removed, generate as many output files as there are input (include) files. The result is a set of formatted files with exactly the same include structure as the input. All the include directives are output and edited to reflect the new filenames. This method of processing include files is indicated by specifying the `-pattern` option on the command line when PasMat is invoked. For further details, refer to the discussion of `-pattern` under "Options" in this appendix.

The default value is `in-`, `in=0` (include files not processed).

`j` Special alignment of declarations and comments. This is a unique numeric directive with the general format

`j=<`*width*`>[±]/<`*col1*`>[sd]/<`*col2*`>c`

<*width*>[±]

Specifies that *width* columns are to be reserved for all following CONST, TYPE, or VAR identifiers (you can also control the alignment of the colons in VAR declarations within the width by using the : option). The optional sign following *width* indicates whether to apply the *width* to record lists (if + is used or the sign is omitted) or to apply it to just the declared variables themselves (if − is specified).

<col1>[sd]

Specifies what column a comment following a statement on the same line is to start in. Note that *width* is a width specification, and *col1* is a column specification. Using *col1* allows you to align all comments in declarations. All comments follow statements (when the comment is the last thing on the same line as the statement), unless you use the options s and d following *col1* (case is ignored and the letters may be in either order). If s is specified, *col1* is only applied to statements and not declarations. If d is specified, then *col1* is only applied to declarations. Omitting both s and d is the same as specifying both; *col1* is applied to all comments following statements if the comment is the last thing on the line.

<col2>c

Specifies a starting column for comments, as *col1* does, but only affects comments that have the trigger character c as the first comment character.

If *width* is omitted, its previous value remains unchanged; the slash in front of the *col1* is required. If *col1* is omitted, the previous value remains unchanged; the slash in front of it is optional unless *col2* is specified, in which case both slashes are required.

For constant declarations, the g=*i* directive (where *i* is greater than one) overrides *width*. Comments should not be used for these statements. Also, the *width* and *col1* values are ignored for a line if they cannot be used because an identifier or its declarative information is too wide. A value of zero for *width*, *col1*, or *col2* disables the corresponding alignment. The default value is j=0/0/0.

k — Indent statements between BEGIN...END pairs. Normally, the statements are indented to the same level as the BEGIN...END pair. The c directive determines the actual placement of BEGIN. Normally, BEGIN appears on a separate line unless c+ is used. Also, k- should be used if c+ is specified. The default value is k-.

l — The case of reserved words and identifiers is to be a literal copy of the input. This directive overrides the w directive and is disabled by the _ directive. The r directive overrides l for reserved words. The default value is l-.

n — Group formal procedure parameters. This is similar to the g+ option, but only for formal parameters of procedure and function declarations. Normally, these appear one per line. The default value is n-.

o — This is a numeric directive (o=*w*) that specifies the output line width. The maximum value allowed is 150 characters. If a particular token will not fit in this width, that line will be lengthened to fit it and a message will be displayed at the end of formatting. The default value is o=80.

q — When IF follows ELSE, do not treat IF specially. It is thus indented on the next line after ELSE. The default value is q-.

r — Output all reserved words in uppercase; otherwise (r-), output in lowercase. The default value is r-.

rec — Indent record field lists under the record's identifier, instead of under the reserved word RECORD. The directive rec+ formats records like this:

```
identifier = RECORD
 {fieldlist}
 END;
```

The default value rec- formats records like this:

```
identifier = RECORD
        {fieldlist}
      END;
```

t — Specifies the amount of tab for each indentation level. This is a numeric directive (t=*n*). Statements that continue on successive lines are additionally indented by half this amount. The -entab command option causes actual tab characters to be placed in the output file using the t directive's tab stop value. The default value is t=2.

u — Case conventions for each identifer are to be based on the identifier's first occurrence in the source. The first occurrence of each identifier is left as is; all subsequent occurrences are made to look exactly like its first occurrence. This option overrides the l and w options, but the e and _ options can still be used. The default value is u-.

v — Align an IF statement so that THEN is indented on the next line after the line containing IF. ELSE is aligned with THEN. The default value is v-.

`w` Convert identifiers to uppercase; otherwise, convert to lowercase. This directive is overridden by the `l`, `e`, and `_` directives. The default value is `w-`.

`x` Suppress space around the arithmetic operators +, −, *, and / and the relational operators =, <>, <, <=, >, and >=. Normally, one space is placed on each side of these operators. This option has no effect on the = used in CONST and TYPE declarations. The default value is `x-`.

`y` Suppress space around the assignment operator :=. The default value is `y-`.

`z` Suppress space after commas. The default value is `z-`.

`@` Controls CASE statement tags (labels). This directive is specified either as a switch (`g+` or `g-`) or as a numeric directive (`@=`*i*). In its `@=`*i* form, the *i* indicates that the statement associated with the CASE tag is to start *i* columns after the start of the case tag. This is similar to the `j=`*<width>/<col1>/<col2>c* directive where *width* indicates how much space to reserve for an identifier being declared. Here the *i* indicates how much space to reserve for the CASE tag(s). If `@=0` (the default), statements following a CASE tag are indented (using the current indenting tab value) on the line following the the tag. If `@=1`, the width of the first tag plus two (for the tag's colon and following space) is used to determine the space to reserve for all following tags in that CASE statement. This means you should put your longest CASE tag first. For `@=`*i* (where *i* is greater than one), *i* spaces are reserved for the CASE tags. If the tag is too wide for the specified width, then the statements that follow are placed on the following line, indented *i* spaces.

`@+` and `@-` specify what to do with a list of tags that don't fit into the specified width. `@+` indicates that a tag that is part of a list is to be put on the next line if it would exceed the *i* width. `@-` indicates that as many tags as possible are to be kept together on the same line. If the resulting list is longer than *i*, the statements are placed on the following line indented by *i*. The default values are `@-`, `@=0`.

: Positioning of colons in aligned VAR declarations. The reserved width for identifiers in declarations is controlled by the `j` directive's *width* parameter. In VAR declarations, you have the choice of allowing the colons to immediately follow their identifiers (by specifying the directive `:-`) or to align the colons at the right end of the reserved width (by specifying the directive `:+`). The default value is `:-`.

\# "Smart" grouping option. If `#+` is specified, then assignment and call statements that were grouped together on the same line in the input will be grouped together on the same line in the output (if they don't exceed the output line width). The default value is `#-`.

_ This directive (an underscore) sets portability mode formatting, which removes the underscore character from identifiers. The first letter of each identifier and the first letter following each underscore character are capitalized, while the remaining characters are lowercase. This directive overrides the `l` and `w` directives. The case of reserved words is set with the `r` directive. The default value is `_-`.

Formatting Comments:

Comments in Pascal are hard to format, and PasMat tries to be clever about it. The rules should allow you to use comments to achieve almost any effect you would like. The following rules govern PasMat's formatting of comments:

- A comment that stands alone on a single line will be passed to the output unaltered. Its left end is set to the current indentation level so that it is aligned with the statements before and/or after it. If it is too long to fit with this alignment, it is placed on the page as far right as it will go.
- A comment that begins as the first thing on a line and continues on another line is passed to the output unaltered, including its indentation. This type of comment is assumed to contain text formatted by the user.
- If a comment covered by one of the above rules will not fit within the defined output line length, the output line is extended as necessary to accommodate it, and a message will be printed at the end of the formatting.

- A comment that is not the first thing on a line is formatted with the rest of the code. Words within it are moved to the next line to make it fit, so nothing that has a fixed format may be used in such a comment. The comment is broken only at blanks; if there is no way to break a comment and still fit the output within the output line length, the line is extended as necessary and a message is written at the end of the formatting. If n · code follows a comment in the input line, then no code will be placed after the comment in the output line. The `j` directive lets you force these comments to start in a specific column. This feature is useful for commenting declarations (see below).

- A comment that follows a statement on a line and begins with a specific character can be forced to start in a specific column. This feature is useful if you are making updates to a program and you want to show who made the update and when.

Statement bunching:

Statement bunching refers to the way PasMat aligns a statement with respect to some component of another statement that precedes it. There are three cases:

- a statement following `CASE` labels
- a statement following `THEN` or `ELSE`
- a statement following `FOR`, `WHILE`, or `WITH`

Because users have diverse styles in formatting these statements, PasMat allows control, to some degree, over how these statements are aligned.

The following discussions on bunching deal with how a statement can be aligned with respect to its "lead-in" statement—that is, whether it's indented after or on the same line as the lead-in. Therefore, *statement* in these cases refers to a simple statement. Compound statements are usually indented starting on a new line (except perhaps for their `BEGIN`s as controlled by the `c` directive).

Bunching with CASE labels: The default formatting rule for a `CASE` statement is to place the selected statements on the same line as the case label(s). The `a` directive lets you make the statement appear on a separate line from the case label. The `@` directive lets you control how far the statements following the case label are indented.

Bunching with IF statements: The default is to place the controlled statements on separate lines. The `b` directive tells PasMat to place the controlled statements on the same line as `THEN` or `ELSE`.

In the special case of ELSE...IF, the default is to put IF on the same line as ELSE. The q directive lets you make IF appear on the next line, indented after ELSE.

Bunching with FOR, WHILE, and WITH: The default is to place the controlled statement on the same line (if it fits). Otherwise, it is indented on the next line. The h directive lets you specify that the statement always appear on the next line.

- *Note:* The h directive also affects the IF statement. With IF bunching off (b- directive) and the h directive off (h-), the controlled statement would normally appear on a separate line. If there is no ELSE, then the h directive applies to the IF statement just like FOR, WHILE, and WITH; that is, the controlled statement is placed on the same line as IF (if it fits).

Tables: Many Pascal programs contain long lists of initialization statements or constant declarations that are logically a single action or declaration. You can fit these into as few lines as possible using the g (g=*i* form) directive. If this is done, tab stops are set up on the line and successive statements or constant declarations are aligned to these tab stops instead of beginning on new lines.

Structured statements, which normally format on more than one line, will not be affected by the g directive. However, care must be taken because assignment and call statements may be grouped with the end of the structured statement (for instance, following an END statement). A special form of grouping directive is provided specifically for assignment and call statements.

Assignment and call statement grouping:

As described above, the grouping directive to format tables is g=*i*, where *i* is the maximum number of statements per line. This sets up tab stops to align up to *i* statements or constant declarations. However, for assignment and call statements it is not always known how many statements will fit on a line. Even if it is known, these statements aligned on tab stops may insert too much space and produce an aesthetically unpleasant result. A special form of grouping can be specified using g+, which affects only assignment and call statements. They are grouped so that as many as possible fit on a line without exceeding the line length. They are never grouped on a line ending a structured statement, so the problem arising with the g=*i* form of grouping cannot happen.

You probably won't want to group all assignment and call statements together everywhere in your program. The preset option is `g-` to format assignment and call statements one per line. Bracket the grouped sections of your code with `g+` and `g-` directives.

If you are formatting a program that is already partially formatted and has sections of code grouped according to the the coder's style, you may not want it reformatted using `g+` and `g-`. The "smart" grouping option (`#+`) lets you specify that if more than one assignment or call statement is on the same input line and they don't exceed the output line width, they will be kept grouped in the output. Thus they will appear in the output exactly as in the input (except perhaps for the space between the statements).

- *Note:* If `g=`*i* is in effect with *i* greater than one, it will have precedence over the effect of `g+` and `#+`. Thus `g+` or `#+` may be enabled and `g=`*i* still be used (except for `g=1`).

Declarations: If you want to align declarations so that the objects of the identifiers (constants or types) all start at a particular column, or align comments explaining the identifiers, use the `j` directive. It allows you to specify the number of columns to reserve for the identifiers and in which column the explaining comment is to begin.

Limitations PasMat has the following limitations:

- The maximum input line length is 255 characters.
- The maximum output line length is 150 characters.
- The input files and output files must be different.
- Only syntactically correct programs, units, blocks, procedures, and statements are formatted. This must be taken into consideration when separate include files and conditional compiler directives are to be formatted.
- The Pascal include directive should be the last thing on the input line if include files are to be processed. Include files are processed to a maximum nesting depth of five. All include files not processed are summarized at the end of formatting. (This assumes, of course, that the `in` directive/option is in effect.)

- The identifiers `Cycle` and `Leave` are treated as reserved Pascal keywords by PasMat. They are treated as two loop control statements by Pascal unless explicitly declared.

- While Pasmat supports Pascal's `$$Shell` facility in `include` files, the processing of MPW's {PInterface} files is *not* fully supported because these files conditionally include files (remember, conditionals are not processed). For this reason, do *not* use the `-in` or `-e` option

Error handling: The following errors are detected and written to diagnostic output:

- In general, premature end-of-file conditions in the input are not reported as errors, to accommodate formatting of individual include files, which may be only program segments. There are cases, however, where the include file is a partial program that PasMat interprets and reports as a syntax error.

- There is a limit on the number of indentation levels that PasMat can handle. If this limit is exceeded, processing will abort. This problem should be exceedingly rare.

- If a comment would require more than the maximum output length (150) to meet the rules given, processing will abort. This problem should be even rarer than indentation level problems.

If a syntax error in the input code causes formatting to abort, an error message will give the input line number on which the error was detected. The error checking is not perfect—successful formatting is no guarantee that the program will compile.

Example

```
PasMat -n -u -r -d -pattern "==formatted/=" Sample.p ∂
"formatted/Sample.p"
```

Format the file Sample.p with the `-n`, `-u`, `-r`, and `-d` options, and write the output to the file "formatted/Sample.p." Include files are processed (`-pattern`) and each Pascal Compiler `$I` include file causes additional output files to be generated. Each of these files is created with the name "formatted/*filename*," where *filename* is the filename specified in the corresponding include file.

Care must be taken when a command line contains quotation marks, slashes, or other special characters that are processed by the Shell itself. This example uses the slash character, so the strings containing it have to be quoted.

The `-pattern` parameter contains a null pattern (`==`) with "`formatted/`" as a replacement string. A null pattern always matches the start of a string.

Appendix I The PasRef Utility

THIS APPENDIX DESCRIBES PASREF, an MPW Shell utility program that you can use to generate a cross-reference table of variable references in your source text.

Syntax

`PasRef` [*option*...] [*sourceFile*...]

Description

Reads Pascal source files, and writes a listing of the source followed by a cross-reference listing of all identifiers. Each identifier is listed in alphabetical order, followed by the line numbers on which it appears. Line numbers can refer to the entire source file, or can be relative to individual include files and units. Each reference indicates whether the identifier is defined, assigned, or simply named (for example, used in an expression).

Identifiers may be up to 63 characters long and are displayed in their entirety unless overridden with the `-x` directive. Identifiers may remain as they appear in the input, or they can be converted to all lowercase (`-l`) or all uppercase (`-u`).

For include files, line numbers are relative to the start of the include file; an additional key number indicates which include file is referred to. A list of each include file processed and its associated key number is displayed before the cross-reference listing.

`USES` declarations can also be processed by PasRef (their associated `$U` *filename* compiler directives are processed as in the Pascal Compiler). These declarations are treated exactly like include files; and, as with the Compiler, only the outermost `USES` declaration is processed (that is, a used unit's `USES` declaration is not processed).

As an alternative to processing `USES` declarations, PasRef accepts multiple source files. Thus you can cross-reference a set of main programs together with the units they use. All the sources are treated like include files for display purposes. In addition, PasRef checks to see whether it has already processed a file (for example, if it appeared twice on the input list or if one of the files already used or included it). If it has already been processed, the file is skipped. ■

Input

If no filenames are specified, standard input is processed. Unless the `-d` option is specified, multiple source files are cross-referenced as a whole, producing a single source listing and a single cross-reference listing. Specifying the `-d` option is the same as executing PasRef individually for each file.

Output

All listings are written to standard output. Form feed characters are placed in the file before each new source listing and its associated cross-reference. Pascal `$P` (page eject) Compiler directives are also processed by PasRef, which may generate additional form feeds in the standard output listing.

Diagnostics

Parameter errors and progress information are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0 Normal termination

1 Parameter or option error

2 Execution terminated

Options

`-a` Process all files even if they are duplicates of ones already processed. The default is to process each (include) file or `USES` unit only *once.*

`-c` Do *not* process a unit if the unit's filename is specified in the list of files to be processed on the command line, or if the unit has already been processed.

`-d` Treat each file specified on the command line as distinct. The default is to treat the *entire* list of files as a whole, producing a single source listing and a single cross-reference listing. This option is the same as executing PasRef individually for each specified file.

`-i` *pathname*`[,` *pathname...*`]`
Search for include or `USES` files in the specified directories. Multiple `-i` options may be specified. At most 15 directories will be searched. The search order is specified under the description of the Pascal command in the *Macintosh Programmer's Workshop 3.0 Reference.*

`-l` Display all identifiers in the cross-reference table in lowercase. This option should not be used if `-u` is specified, but if it is, the `-u` is ignored.

`-ni | -noincludes`
Do not process the include files. The default is to process the include files.

`-nl | -nolisting`
Do not display the input source as it is being processed. The default is to list the input.

`-nolex` Do not display the lexical information on the source listing. See the example at the end of this appendix for further details.

`-nt | -nototal`
Do not display the total line count in the source listing. This option is ignored if no listing is being generated (`-nl`).

`-n[u] | -nouses`
Do not process `USES` declarations. The default is to process `USES` declarations. If `-nu` is specified, the `-c` option is ignored.

`-o` The source file is an Object Pascal program. The identifier `OBJECT` is considered to be a reserved word so that Object Pascal declarations may be processed. The default is to assume the source is not an Object Pascal program.

`-p` Display version and progress information on the diagnostic file.

`-s` Do not display include and `USES` information in the listing or cross-reference, and cross-reference by total source line number count rather than by include-file line number.

`-t` Cross-reference by total source line number count rather than by include-file line number. This option can be used if you are not interested in knowing the positions in included files. However, the include information is still displayed (unless `-s`, `-ni`, or `-nu` is specified). This option is implied by the `-s` option.

`-u` Display all identifiers in the cross-reference table in uppercase. This option should not be used if `-l` is specified.

`-w` *width* Set the maximum output width of the cross-reference listing. This setting determines how many line numbers are displayed on one line of the cross-reference listing. It does not affect the source listing. *Width* can be a value from 40 to 255; the default is 110.

`-x` *width* Set the maximum display width for identifiers in the cross-reference listing. (The default is to set the width to the size of the largest identifier cross-referenced.) If an identifier is too long to fit in the specified width, it is indicated by preceding the last four characters with an ellipsis (...). *Width* can be a value from 8 to 63.

Normally, both include files and USES declarations are processed. The `-noincludes` option suppresses processing of include files. The `-nouses` option suppresses processing of USES declarations.

Omitting the `-nouses` option causes PasRef to process a USES declaration exactly as does the Pascal Compiler. However, you may want to cross-reference an entire system, including all the units of that system. Processing the units through the USES declaration would cause only the interface section of each unit to be processed. If the `-nouses` option is used, then USES will not be processed and each unit from the parameter list can be cross-referenced, treating the entire list as a single source.

PasRef can also cross-reference all the units of a program while still expanding other units not directly part of that program, such as the Toolbox units. In that case, the `-c` option should be used. With the `-c` option, if the ($U interface) filename is the same as one of the filenames specified on the parameter list, then the unit will not be processed from the USES declaration because its full source will be (or has been) processed.

To summarize, you have the following choices:

- Don't process the USES, and specify a list of all files you want to process, by using the `-nouses` option.
- Process only the interface sections through the USES declarations (like the Compiler), by omitting the `-nouses` option.
- Process some of the units through the USES and others as full sources, by specifying the `-c` option.

In all cases where a list of files is specified, no unit will ever be processed more than *once* (unless the `-a` option is given).

Example `PasRef -nu -w 80 Memory.p > Memory.p.Xref`

Cross-reference the sample desk accessory Memory.p and write the output to the file Memory.p.Xref. No `USES` are processed (`-nu`). The following source and cross-reference listings are generated:

```
  1   1    1 --       {
  2   1    2 --        File Memory.p
  3   1    3 --
  4   1    4 --        Copyright Apple Computer, Inc. 1985-1987
  5   1    5 --        All rights reserved.
  6   1    6 --       }
  7   1    7 --
  8   1    8 --        {$D+} { MacsBug symbols on }
  9   1    9 --        {$R-} { No range checking }
 10   1   10 --
 11   1   11 --        UNIT Memory;
 12   1   12 --
 13   1   13 --        INTERFACE
 14   1   14 --
 15   1   15 --        USES
 16   1   16 --           MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
 17   1   17 --
 18   1   18 --
 19   1   19 --       FUNCTION DRVROpen         (ctlPB: ParmBlkPtr; dCtl: DCtlPtr): OSErr;
 20   1   20 --       FUNCTION DRVRControl      (ctlPB: ParmBlkPtr; dCtl: DCtlPtr): OSErr;
 21   1   21 --       FUNCTION DRVRStatus       (ctlPB: ParmBlkPtr; dCtl: DCtlPtr): OSErr;
 22   1   22 --       FUNCTION DRVRPrime        (ctlPB: ParmBlkPtr; dCtl: DCtlPtr): OSErr;
 23   1   23 --       FUNCTION DRVRClose        (ctlPB: ParmBlkPtr; dCtl: DCtlPtr): OSErr;
 24   1   24 --
 25   1   25 --
 26   1   26 --       IMPLEMENTATION
etc.
 63   1   63 -- A     FUNCTION DRVRClose(ctlPB: ParmBlkPtr; dCtl: DCtlPtr): OSErr;
 64   1   64 0- A     BEGIN
 65   1   65 --           IF dCtl^.dCtlwindow <> NIL THEN
 66   1   66 1-           BEGIN
 67   1   67 --                DisposeWindow (WindowPtr(dCtl^.dCtlWindow));
 68   1   68 --                dCtl^.dCtlWindow := NIL;
 69   1   69 -1           END;
 70   1   70 --           DRVRClose := NOErr;
 71   1   71 -0       A END;
etc.
178   1  178 --
179   1  179 --       END. {of memory UNIT}
180   1  180 --
```

Each line of the source listing is preceded by five columns of information:

- The total line count.
- The include key assigned by PasRef for an include or USES file (see below).
- The line number within the include or main file.
- Two indicators (left and right) that reflect the static block nesting level. The left indicator is incremented (mod 10) and displayed whenever a BEGIN, REPEAT, or CASE is encountered. On termination of these structures with an END or UNTIL, the right indicator is displayed, then decremented. It is thus easy to match BEGIN, REPEAT, and CASE statements with their matching terminations.
- A letter that reflects the static level of procedures. The character is updated for each procedure nest level (A for level 1, B for level 2, and so on), and displayed on the line containing the heading and on the BEGIN and END associated with the procedure body. Using this column, you can easily find the procedure body for a procedure heading when there are nested procedures declared between the heading and its body.

The cross-reference listing follows:

```
 1. Memory.p

-A-
 accEvent    144 ( 1)
 accRun     158 ( 1)
 ApplicZone  121 ( 1)
 Away      33*( 1) 146 ( 1)

-B-
 BeginUpdate 151 ( 1)
 BNOT         39 ( 1)
 Bold         90 ( 1) 117 ( 1)
 Boolean      31*( 1)
 BOR          39 ( 1)
 BSL          39 ( 1)

-C-
 csCode      143 ( 1)
 CSParam     146 ( 1)
 ctlPB        19*( 1)  20*( 1)  21*( 1)  22*( 1)  23*( 1)  43*( 1)
              63*( 1)  74*( 1) 143 ( 1) 146 ( 1) 168*( 1) 173*( 1)
```

```
-D-
 dCtl          19*( 1)  20*( 1)  21*( 1)  22*( 1)  23*( 1)  37*( 1)
               39 ( 1)  43*( 1)  50 ( 1)  53 ( 1)  54 ( 1)  55 ( 1)
               63*( 1)  65 ( 1)  67 ( 1)  68 ( 1)  74*( 1) 115 ( 1)
              142 ( 1) 168*( 1) 173*( 1)
 DCtlPtr       19 ( 1)  20 ( 1)  21 ( 1)  22 ( 1)  23 ( 1)  37 ( 1)
               43 ( 1)  63 ( 1)  74 ( 1) 168 ( 1) 173 ( 1)
 dCtlRefNum    39 ( 1)  54 ( 1)
 dCtlWindow    50 ( 1)  55=( 1)  67 ( 1)  68=( 1) 142 ( 1)
etc.
-V-
 VolName       79*( 1) 100 ( 1) 135 ( 1)

-W-
 what         149 ( 1)
 WindowKind    54=( 1)
 windowpeek    54 ( 1)
 WindowPtr     48 ( 1)  67 ( 1) 151 ( 1) 153 ( 1)
 wRect         47*( 1)

*** End PasRef: 105 id's  249 references
```

The numbers in parentheses following the line numbers are the include keys of the associated include files (shown in column 2 of the source listing). The include-file names are shown following the source listing. Thus you can see what line number was in which include file. An asterisk (*) following a line number indicates a definition of the variable. An equal sign (=) indicates an assignment. A line number with nothing following it means a reference to the identifier.

Limitations

PasRef does not process conditional compilation directives! Thus, given the "right" combination of `$IFC`s and `$ELSEC`s, PasRef's lexical (nesting) information can be thrown off. If this happens or if you just don't want the lexical information, you may specify the `-nolex` option.

PasRef stores all its information on the Pascal heap. Up to 5000 identifiers can be handled, but more identifiers will mean less cross-reference space. A message is given if PasRef runs out of heap space.

- *Note:* Although PasRef never misses a reference, it can infrequently be fooled into thinking that a variable is defined when it actually isn't. One case where this happens is in record structure variants. The record variant's case tag is always flagged as a definition (even when there is no tag type), and the variant's case label constants (if they are identifiers) are also sometimes incorrectly flagged depending on the context. (This occurs only in the declaration parts of the program.)

While PasRef supports Pascal's `$$Shell` facility in `include` files and `USES` declarations, the processing of MPW's {PInterfaces} files is *not* fully supported because these files conditionally include files (remember, conditionals are not processed). For this reason, always use the **-nu** option to suppress processing of `USES` declarations.

The identifiers `Cycle` and `Leave` are treated as reserved Pascal keywords by PasRef. These are treated as two loop control statements by Pascal unless explicitly declared.

Appendix J **The ProcNames Utility**

THIS APPENDIX DESCRIBES ProcNames, an MPW Shell utility program that you can use to produce a list of all procedure and function names in your Pascal program or unit.

Syntax

`ProcNames` [*option*...][*file*...]

Description

Accepts a Pascal program or unit as input and produces a listing of all its procedure and function names. The names are shown indented as a function of their nesting level. The nesting level and line number information is also displayed.

ProcNames can be used in conjunction with the Pascal "pretty-printer" PasMat when that utility is used to format separate include files. For that case, PasMat requires that the initial indenting level be specified. This level is exactly the information provided by ProcNames.

The line number information displayed by ProcNames exactly matches that produced by the Pascal cross-reference utility PasRef (with or without `USES` being processed), so ProcNames may be used in conjunction with the listing produced by PasRef to show just the line numbers of every procedure or function header.

Another possible use for the ProcNames output is to use the line number and file information to find procedures and functions quickly with Shell editing commands.

Input

The file parameters specify a list of Pascal source filenames to be processed. Standard input is processed if no filenames are specified. Unless the `-d` option is specified, the entire list of files is treated as a single group of files to be processed as a whole, producing a single procedure or function summary. Specifying the `-d` option is equivalent to executing ProcNames individually for each specified file.

Output

The procedure or function name listing is written to the standard output file. Form feed characters are placed in the file before each new list (unless the `-e` option is specified). ■

Diagnostics Errors are written to the diagnostic file.

Status The following status codes may be returned to the Shell:

0 Normal termination

1 Parameter or option error

2 Execution terminated

Options

`-c` Do *not* process a used unit if the unit's `$U` interface filename is specified in the list of files to be processed. This option has the same effect on the line numbering as does the `-c` option in the PasRef utility.

`-d` Reset total line number count to 1 on each new file. If a list of files is specified, then the total line number count may either start at 1 or continue from where it left off in the previous file. The default is to agree with the listing produced by PasRef when it processes a list of files, that is, to *continue* the count. However, if you want ProcNames to treat each file independently, you may specify the `-d` option so that the total line number count is reset to 1 before each file is processed.

`-e` Suppress page eject (form feed) between each procedure or function listing.

`-f` PasMat format compatibility mode. The default lists the procedure and function names as a function of their Pascal compiler indenting level. However, for indenting purposes only, a special case is made of level-1 procedures in the `IMPLEMENTATION` section of a unit. PasMat formats these procedures indented under the word `IMPLEMENTATION`. Thus they are indented as if they were level-2 procedures. If you intend to use the level information for PasMat, then you should specify the `-f` option.

`-i` *pathname*[, *pathname*...]
Search for include or `USES` files in the specified directories. Multiple `-i` options may be specified. At most, 15 directories will be searched. The search order is specified under the description of the `-i` option for the Pascal command.

`-n` Suppress all line number and level information in the output display. Only the procedure and function names will be shown appropriately indented.

`-o` The source file is an Object Pascal program. The identifier `OBJECT` is considered as a reserved word so that Object Pascal declarations may be processed. The default assumes that the source is an Object Pascal program.

-p Display version information and progress information on the diagnostic file.

-u Process USES declarations. The only reason you would need to process USES with ProcNames would be to make the line number information agree with a PasRef listing that also contains processed USES. The default does not process the USES declarations because they have no effect on the procedure name listing, only the associated line numbers. Thus, if you specify the -n option to suppress the line number information, it makes no sense to process USES, so the -u option will be ignored when the -n option is specified.

Examples

```
ProcNames Memory.p >names
```

List all the procedures and functions for the Pascal program Memory.p and write the output to the filenames. The listing below is the output generated in the names file:

```
Procedure/Function names for Memory.p

 11  11  0   Memory    [Main]     Memory.p
 37  37  1   RsrcID
 43  43  1   DRVROpen
 63  63  1   DRVRClose
 74  74  1   DRVRControl
 76  76  2   DrawWindow
 83  83  3   PrintNum
 93  93  3   GetVolStuff
108 108  3   PrtRsrcStr
168 168  1   DRVRPrime
173 173  1   DRVRStatus

*** End ProcNames: 11 Procedures and Functions
```

The first two columns on each line are line number information. The third column is the level number. The first column shows the line number of a routine within the total source. The second column shows the line number within an include file (includes are always processed). As each include file changes, the name of the file from which input is being processed is shown along with the routine name on the first line after the change in source. Segment names (from Pascal Compiler $S directives) are similarly processed. These are shown enclosed in square brackets (the blank segment name is shown as "[Main]").

Limitations

Only syntactically correct programs are accepted by ProcNames. Conditional compilation Compiler directives are *not* processed.

Appendix K Advanced Topics for 68020 Programmers

THIS APPENDIX SUMMARIZES THE SUPPORT MPW PASCAL PROVIDES for the Motorola 68020 central processing unit. In addition, it considers some programming implications such as longint arithmetic and bit-field operations. ■

Contents

Support for the 68020

MPW Pascal provides support for the 68020 central processing unit in the following ways:

- The -MC68020 Compiler option or the equivalent {$MC68020+} Compiler directive permits the Compiler to generate 68020 instructions: CHK.L, CHK2, RTD MULS.L, DIVS.L, BFFXXX, and EXTB.L.
- The use of the 68020 instructions has two main advantages: faster longint arithmetic and more efficient use of packed data types.
- If you elect to use the -MC68020 option, your application may run only if a 68020 is present in your Macintosh.

Faster longint arithmetic

The 68020 provides new instructions for longint multiplication and division. The -MC68020 option permits the Compiler to generate these instructions.

Bit-field operations

The -MC68020 option permits the Compiler to generate the 68020 bit-field instructions. These instructions can significantly improve performance in the use of packed data types.

Glossary

access: To use a variable's identifier in source text

actual parameter: A parameter whose value is given to a formal parameter by a procedure or function call.

address: A number that specifies a location in memory.

allocate: To reserve an area of memory for use.

ancestor: The object from which another object is created.

application: a program that can be run under the Macintosh Finder or Multifinder.

array: A data structure containing an ordered set of elements.

ASCII: Acronym for *American Standard Code for Information Interchange*, a system of assigning code numbers to letters, numerals, punctuation marks, and control codes.

assignment compatible: Of two types, capable of being combined in an assignment statement.

associated scalar type: The type of the elements of a subrange.

base type: The type of the members of a set.

blank: A tab, space, return, or Option-space character.

block: The fundamental large-scale unit of a Pascal program.

Boolean expression: An expression whose value is either `true` or `false`.

comment: Source text intended for a human reader, ignored by the Compiler.

Compiler directive: A symbol placed in Pascal source text to send an instruction to the Compiler.

Compiler option: A symbol placed in the MPW Pascal command line to send an instruction to the Compiler.

compile-time expression: An expression whose value controls conditional compilation.

compile-time variable: A variable, created by the `$SETC` directive, that goes into a compile-time expression.

component type: The type of the elements of a structured type.

constant: An identifier that represents a fixed, unchanging value.

constant declaration part: A part of a block that contains constant declarations.

current file position: The position in a file currently accessed by the file window variable.

defining declaration: The block of a forward declaration.

delimiter: A symbol that separates other symbols in source text.

descendant: An object created by another object.

desk accessory: a program that you run by selecting it from the Apple menu.

diagnostic output file: A file (often the topmost window) to which the Compiler sends error messages and information about its progress.

digits: The numerals 0..9.

dimension: An ordering relation among elements of an array.

directive: A source text symbol that modifies the action of the Compiler.

dynamic variable: A variable created during program execution.

exception: An unusual condition arising during execution of an instruction. Exceptions can also be externally generated, for example, interruptsbus errors, or reset.

expression: Any representation of a value. It can be a single identifier of a constant or variable, or a combination of identifiers and operators.

external file: A peripheral device or disk file that contains the value of a file variable.

factor: A part of a term.

field: A data structure within a record or object.

file window variable: A buffer variable that accesses one component of a file at a time.

fixed-point number: A signed 32-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word.

Floating-Point Arithmetic Package: A Macintosh package that supports extended-precision arithmetic according to IEEE standard 754.

floating-point: A way of representing decimal numbers.

floating-point coprocessor (MC68881): A coprocessor chip that provides high-speed support for extended-precision arithmetic.

formal parameter: A parameter in a procedure or function declaration.

forward declaration: A procedure or function declaration whose block occurs later in the source text.

free block: A memory block containing space available for allocation.

global scope: The scope of code or data that is accessible throughout a program.

handle: A pointer to a master pointer, which designates a relocatable block in the heap by double indirection.

hexadecimal: Base-16 number representation, using the numerals 0..9 and the letters A..F.

hex digits: Symbols representing the hexadecimal numerals.

host program: A program or unit that uses a unit.

identified variable: A variable pointed to by a pointer.

identifier: A name in source text.

implementation part: The part of a unit containing code that executes the procedures and functions declared in the interface part.

implicit parameter: An undeclared parameter of a method, such as `Self`.

index: A numeric value that indicates the position of an element in a sublist or array, expressed by a subscript.

index type: The type of an index expression.

inheritance: The process by which one object generates another object.

input: The process of entering data into an executing program.

interface file: A code file that provides an interface between a specific language and a library.

interface part: The part of a unit that is available to a host program.

interrupt: An exception that's signaled to the processor by a device to notify the processor of a change in condition of the device, such as the completion of an I/O request.

interrupt handler: A routine that services interrupts.

I/O: Abbreviation for *input* and *output* operations, taken collectively.

label: A name that identifies a location in source text.

label declaration part: A part of a block that contains label declarations.

length: Of a string, the number of characters in its actual value.

letters: The symbols A..Z and a..z.

library: A code file that contains procedures and functions available to a program.

local scope: The scope of code or data that is accessible in only part of a source text.

logical record: A component of a file.

Macintosh Programmer's Workshop (MPW): Apple's software development environment for the Macintosh family.

master pointer: A single pointer to a relocatable blcok, maintained by the Memory Manager and updated whenever the block is moved, purged, or reallocated. All handles to a relocatable block refer to it by double indirection.

member: The relation of an object to its type.

memory block: An area of contiguous memory within a heap zone.

method: A procedure or function in an object.

method call: A special type of function call that calls a method in an object.

MPW Shell: The application that provides the environment within which the other parts of the Macintosh Programmer's Workshop operate. The Shell combines an editor, command interpreter, and built-in commands.

MPW tool: An executable program (type `'MPST'`) that is integrated with the MPW Shell environment (contrasted with an application, which runs
stand -alone). Examples of MPW tools are ProcNames, PasRef, PasMat, and the Pascal Performance Tools.

mutual recursion: The situation in which two or more procedures and/or functions call each other.

NaN: Acronym for *Not a Number*, the result of a meaningless arithmetical operation.

null string: A string of zero length, containing no characters.

object: A program structure that contains both data (called *fields*) and routines (called *methods*).

Object Pascal: An extension of Pascal based on the use of objects.

object reference variable: A variable declared with an object type.

object type: The type of an object.

operand: Data that controls or modifies the action of an operation.

operator: A symbol that acts upon one or two operands, generating a new value.

output: The process of accessing data generated by an executing program.

package: A set of routines and that types that's stored as a resource and brought into memory only when needed.

predefined: Of an identifier, having its meaning supplied by the Compiler.

procedure and function declaration part: The part of a block that contains procedure and function declarations.

program: A complete, executable Pascal source text.

qualifier: A symbol that modifies a variable access.

quoted string constant: A sequence of ASCII characters enclosed in apostrophes.

relational operator: An operator that compares two operands, producing a `boolean` result. Relational operators are listed in Table 6-6.

reserved word: A word or sequence of characters reserved by Pascal for special use, and therefore unavailable as an identifier in a Pascal program.

scope: The area of source text in which an identifier can be referenced.

segment: A part of code that can be separately loaded into memory.

set constructor: An expression enclosed in brackets that defines a set.

short-circuit operator: An operator that evaluates two operands from left to right, and does not evaluate the second if the first produces a true result.

simple expression: A combination of a term with a sign, OR, or |.

simple type: A real type, scalar type, or string type.

source text: Text written by a programmer.

special symbol: A punctuation mark recognized by the Compiler.

stack: The area of memory in which space is allocated and released in LIFO (last-in, first-out) order.

stack frame: The area of the stack used by a routine for its parameters, return address, local variables, and temporary storage.

standard input: The input that the MPW Shell gives to Pascal input operations by default.

standard output: The output that the MPW Shell gives to Pascal output operations by default.

statement part: A part of a block that contains statements.

structured type: A data type that stores more than one value.

subscript: A numeric expression whose value is the index of an element in a string or an array.

symbol: A lexical component of source text processed by the Compiler.

tag field: A field of a record that contains information used to identify variant fields.

tag field identifier: The identifier used to access a tag field.

term: A part of an expression.

Transcendental Functions Package: A Macintosh package that contains trigonometric, logarithmic, exponential, and financial functions as well as a random number generator.

trap: An exception caused by instruction execution. It arises from either process recognition of abnormal conditions during instruction execution or from use of the specific instructions whose normal behavior is to cause an exception.

type: The kind of quantity represented by a data value.

type declaration: Pascal source text that associates an identifier with a type.

type declaration part: A part of a block that contains type declarations.

underscore: The symbol _ (ASCII 95).

unit: A separately compiled collection of types, variables, procedures, and/or functions that are not executable by themselves but may be used by a program.

user-defined: Of an identifier, requiring a meaning to be supplied by a program.

user-defined anonymous type: A user-defined type that does not have an identifier.

variable: A symbol that represents a location in memory where a value can be stored.

variable declaration part: A part of a block that contains variable declarations.

variant: A group of fields that share memory space with other fields.

Index

N

O

P

THE APPLE PUBLISHING SYSTEM

This Apple® manual was written, edited, and composed on a desktop publishing system using Apple® Macintosh® computers and Microsoft® Word software. Proof and final pages were created on the Apple LaserWriter® IINTX printer. POSTSCRIPT®, the LaserWriter® page-description language was developed by Adobe Systems Incorporated. The illustrations were created using Adobe Illustrator and some were output to a Linotronic 300.

The illustration on the cover was generated using Adobe Illustrator 88 on a Macintosh® II computer. Some of the images were scanned using an Apple® Scanner and then manipulated in ImageStudio. Initial proofing was done using a QMS color printer. Color separations were done using Adobe separator and output to a Linotronic 300 at standard resolution.

Text type is Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats®. Some elements, such as programs listings, are set in Apple Courier, a fixed-width font.