WORKING PAPER 82

UNDERSTANDING LISP PROGRAMS:

TOWARDS A PROGRAMMER'S APPRENTICE

Charles Rich and Howard E. Shrobe

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

December, 1974

Abstract

Several attempts have been made to produce tools which will help the programmer of complex computer systems. A new approach is proposed which integrates the *programmer's intentions*, the program code, and the comments, by relating them to a *knowledge base* of programming techniques. Our research will extend the work of Sussman, Goldstein, and Hewitt on *program description* and *annotation*. A prototype system will be implemented which answers questions and detects bugs in simple LISP programs.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-70-A-0362-0005.

TABLE OF CONTENTS

CHAPTER ZERO. MOTIVATION AND SCENARIO

0.0 The Complexity Barrier

During the past decade the power of computational facilities has increased by several orders of magnitude. The transition from tab equipment systems to the modern day computer utility, exemplified by *MULTICS*, has taken little more than two decades. Moreover we are faced with the realistic prospect that current generation hardware will itself be superseeded within another decade by *LSI* and other technologies, sophisticated enough to house in a desk drawer computers more powerful than those of the last generation.

During this period, software develoment has also proceeded at an amazing rate. It has similarly taken only about two decades for the transition from the first *FORTRAN* compiler, to modern *PL/I* (and to other structured languages), optimizing compilers, sophisticated data-base managing systems, complex operating systems like *MULTICS*, etc. Within the Artificial Intelligence community, this same progress has taken place. The transition from the batch *LISP* 1.5 to higher powered interactive dialects such as *MACLISP* and *INTERLISP* has also taken less than two decades. In addition, new specialized A.I. languages have been developed, e.g. *PLANNER, CONNIVER, and QA4.*

Unfortunately, the result of such advances has been, to a large extent, merely to open Pandora's box. Each advance in computing hardware or in the power of programming languages, has spawned a new generation of yet more sophisticated and complex programs. Modern large scale programs are, to a large extent, caught on the horns of a dilemma. On the one hand, the sheer magnitude of most large software systems dictates that they be produced in a project in which

responsibilities for sub-modules is parcelled out; on the other hand, the interactions between these submodules are frequently so diverse that they defy coordination among an entire project of individuals none of whom see the entire picture because of this division of labor. This suggests that design and even coding must be accomplished by a single individual. Unfortunately, one individual usually is no more successful at keeping all the interactions straight than is the entire project. The end result of this situation is that software is both notoriously late and famously unreliable.

There are additional difficulties brought about by the economics of large scale production. Programmers have become "proletarianized". The elite expert programmer who crafted a system and stayed with it for many years, finely tuning it and adding new bells and whistles with ease, has by and large been superseeded by an entire generation of college graduates who were introduced to computing in their courses, and who are hired and fired by programming shops in accord with the winds of the market place. Each such individual must pay the price of getting up to speed on the current system being produced, usually mastering only that corner of the system necessary for his individual task before he is transferred to another project or layed off. The net result of this process is that each new feature added to a system carries with it an extreme likelihood of introducing a new "bug". The computer software industry has a folklore of "horror stories" caused by this process. Time-sharing systems are put on the air only to crash seven times in the first hour, deleting some user's files in the process; companies switch to a canned inventory system only to find out that they no longer know how many of certain items they now have, etc.

We have, thus, come up against what Terry Winograd has referred to as the "complexity barrier". Winograd, working not in large scale commercial programming, but in the research environment

of the M.I.T. Artificial Intelligence Laboratory, observed this same phenomenon of programs growing larger and more complex than could be handled by either an individual or a project. *SHRDLU*, Winograd's *magnum opus*, is precisely such a program. Designed as a research project in computer understanding of natural language, *SHRDLU* also incorporates a problem solving component to solve extremely simple construction tasks in a world which contained a collection of toy blocks, boxes and a table. Even after several rounds of "cleaning up the code" (which had the express purpose of clarifying the interactions), it is still well known that parts of the program (particularly those which involve interaction between the semantic specialists and the dictionary) should be touched only by a select crew of experts. Given that *SHRDLU*, impressive as it is, is not anywhere near to the machine intelligence to which A.I. aspires, Winograd and others (including the authors of this paper) have concluded that continued research in A.I. is dependent on producing a means of breaking through this barrier of complexity.

It should be realized that this barrier is not caused simply by the size of the program, but rather is due to the fact that, as the size increases, the number of relationships between modules (assuming the code is coded modularly) increases considerably quicker. In order for a program to work, it is necessary for these interactions (function or subroutine calls, shared variables, etc.) to be both syntactically, and semantically correct. For example, a routine might expect as input a particular type of *list* called an *s-marker list*. A syntactic check, at least in *LISP*, could only verify that the routine is being passed a *list*; a semantic check would verify that a *s-marker list* was being passed in. The essense of the complexity barrier is that, as the size of the program grows, the much more rapid growth of the interactions between modules makes it virtually impossible to design a new module which can function within the constraints of the existing program.

0.1 RELATIONSHIP TO OTHER WORK

0.1.1 Limitations of Previous Approaches

A great deal of the work done within Computer Science departments for the past several years has to some extent or other been motivated by a desire to deal with this issue of bringing the complexity problem under control. Several approaches have been proposed, all of which suggest that the answer is to force the computer to help manage the complexity.

A first cut solution to the complexity problem involves the construction of a set of tools to ease the programmer's job. Such tools would include cross reference generators, pretty-printers, various break-point setters and related debugging aids, etc. The best example of this approach is the work done in constructing *INTERLISP* <Teitelman, 1974>. What typifies this approach, is the idea that the system should provide all the assistance that it can, without its having to know very much about the program at hand.

The limitation of this approach is that a system so designed can provide little aid to the programmer in actually designing the program. This is intrinsic to the approach; the system is not expected to have knowledge of the programmer's intentions to any real degree. The services rendered by this type of system are limited to a very valuable collection of essentially syntactic aids. Such aids provide extremely valuable information: cross references, stack snapshots on failure, etc.; all of these are indispensible to a programmer trying to design or debug a program. They do require, however, that the programmer actively knows the interactions, goal structure and overall intentions of the whole system (or at least of that part of it on which he is currently working). Keeping track of these facts tends to be a task which is better suited to a machine than a person,

because it exceeds the storage capacities of most people's active memory. To summarize, these tools provide much needed help, but they do fall short of breaking the complexity barrier.

A second approach has been to design new languages or formalisms which in some way will bring the complexity problem under control by imposing structure on the code. This general trend has come to be called *structured programming*, and in the non-A.I. world is most associated with Edsger Dijkstra, C.A.R. Hoare, and others <Dahl, Dijkstra, and Hoare, 1972>. this school has also advocated changes in the methodology of programming, the central ideas being "top down programming", "stepwise refinement", "goto-less programming", and "modularity".

Within the Artificial Intelligence community there have also been a number of researchers involved in the development of better languages and formalisms. In particular, there has been what seems to be a never ending series of powerful new languages, each claiming to solve many of the problems of writing large A.I. systems (and each suceeding to some extent). Such efforts include PLANNER <Hewitt, 1971>, CONNIVER <Sussman and McDermott, 1972>, QA4 <Rulifson, Dirksen, and Waldinger, 1972>, and the ACTORS formalism <Hewitt, et. al., 1973>. Although all of these researchers would not consider themselves part of the structured programming movement, (quite the contrary, most of these languages are designed to escape, in one way or another, the rules of "structured programming"), there is still the shared assumption (with which we agree) that better formalisms and languages can solve some of the complexity problem.

Formalism and language design does promise to provide us with techniques which will help develop clearer, more structured programs. Yet the problem we face can only be partly solved through this approach. For one thing, there is a well known phenomenon that stronger formalisms

breed more complex programs; e.g. *PLANNER* and its decendents have opened A.I. programming to the type of complexity exemplified by *SHRDLU*. This difficulty, however, is not really the crux of the matter; after all, we really do want to construct programs as complex as this (in fact, much more complex). The problem is that, within the realm of commercial programming as well as within the A.I. community, it is unrealistic to think that even the perfect language could be adopted overnight. Old programs have to be maintained, new language processors need to be implemented on a host of different machines, and most importantly, programmers have to adopt to the new language or methodology. This simply does not happen quickly, if at all; most programmers still use *FORTRAN* or *COBOL*; most A.I. programming is still done in *LISP*. Thus, we feel that it is more productive at this point to discuss means of helping programmers with the complexities they encounter within the language systems they currently use; nevertheless, we feel it important to note that the system which we will propose constructing in this document will be capable of a rather straightforward adaptation to new languages.

A radical approach to the complexity problem has been to suggest that the easiest way out is simply to make the machine do everything; i.e. *automatic programming*. This approach has been put forward most clearly by Robert Balzer <"Automatic Programming", Balzer 1973>, and is currently being investigated by several researchers at M.I.T.. The proposed idea is to have the machine produce efficient code, given only an English or some other "high level" description of the problem. Although, this approach does seem seductive, it is our estimate that it will not in the short run produce results of much value to the designer of large-scale A.I. programs (or other large scale programs). It is interesting to note, however, that our approach does seem in many ways to be a step towards such "automagic programming".

Another criticism of automatic programming is that, in general, computers ought to *aid* in programming, not assume the overall task. This can be seen by looking at applications of computers in engineering disciplines other than software engineering. At present, although computers play a valuable role as aids in architectural design, it is unclear that the aesthetic sense necessary for automatic design could be formalized to the extent necessary to package it into computer programs. (It is, after all, quite difficult to package it into people). Furthermore, the problem of representation of vague concepts, is precisely the crux of real attempts to simulate human intelligence on the computer. This suggests that in the near future automatic programming ought not to, and will not be of appreciable help in constructing the type of complex programs which typify work in A.I.

A final idea with much currency is that of *program vertfication.* The central idea of this approach is to construct in the first order logic a statement of some property of the program (usually an overall statement of the program's behavior). Further, it is observed that various other statements in the first order logic can be attached to locations in the *flowchart* of the program. These are then used to construct a *proof* of the desired property of the program. Most well known among the advocates of this approach are Floyd <"Assigning Meaning to Programs", 1967>, Burstall <"Proving Properties of Programs by Structural Induction", 1969>, and others. Peter Deutsch <"An Interactive Program Verifier", 1973> has constructed a program which performs this verification function for a series of numerical programs of moderate complexity. Although he departs in several respects from the approach originally presented by Floyd, it is still fair to say that the approach basically involves a resolution-like theorem-prover working on assertions in the first order logic. Because of this a sizable part of his effort is diverted into keeping the theorm proving process from engaging in exponential explosions.

In general, this approach seems to us to have two difficulties. First, the process is best suited to proving programs correct once they have been designed, whereas we see the main problem as designing programs within a highly complex domain. Secondly, although we find the idea of certifying programs to be attractive, the method used by the above researchers forces the programmer to express his intentions in a language (first order logic) which is frequently unnatural to him, and in some cases inadequate for the task. Furthermore, the knowledge used in constructing these proofs is itself often obscure (for example, Burstall <Burstall, 1972> uses category theory to prove properties about programs using list structure). Because of this, the system would be inaccessible to the average programmer who has a more "common-sense" understanding of his program design.

We have summarized these approaches not to take "cheap shots" at them, but rather to see what limitations they have run into. Chief among these difficulties is the inability to bring into use the basic knowledge of programming skills which the average programmer has at his disposal. Furthermore, all of these approaches have difficulty integrating into their operation the programmer's knowledge of the overall intentions and goal structure of his program. They, therefore, have to either explicitly disavow certain types of services or to remove the programmer from the formulation of the program's design.

Although there is great value to many of the services which these various approaches can render, we feel that they suffer from not being integrated into the proper total system. In the next two sections we will describe and present scenarios of what we think such a total integrated system ought to be like. Many of the capabilities of these earlier approaches will be incorporated into the system which we are proposing. Nevertheless, it would be incorrect to include our system in any of

the previously reviewed categories.

0.1.2 Knowledge Based Approaches

Within the Aritificial Intelligence Laboratory at M.I.T. there has developed over the past several years a growing belief that the essense of building intelligent automata is contained within the question of how to build a *knowlege based* system which can employ its base of knowledge to solve problems within a particular domain. We propose to try to apply this approach to the problem domain of program design, verification, documentation and bug detection.

The overall motivation for this project is the belief that man-machine interaction can be a symbiotic relationship in which the overall productivity is greater than the sum of the parts. This is a large question which could be approached in any of a number of engineering disciplines. We have chosen software engineering for several reasons. First, we know it best. Secondly, it presents an area of large complexity where we can both break out of the traps accompanying "toy problems" and still cut the domain down to a manageable size. Finally, and perhaps most significantly, software engineering is very much a case of the cobbler's child who has no shoes. We hope to at least save a sole in this thesis.

We, therefore, are not intending to begin research on how to replace programmers, but rather on how a knowledgeable computer could help an already competent programmer. It has been our experience that we can produce better and cleaner code faster when working with a partner who shares our understanding of the intentions and goal structure of our program. We, therefore, believe that the appropriate metaphor for our work is that of creating a program with the capabilities of a junior colleague working on a joint project. The program should know the

problem domain, implementation techniques, and the programming language being used fairly well. It need not know everything in advance; it can always ask its senior partner for advice or further information. Furthermore, this program might well be capable of paying more attention to details, of writing trivial parts of the code, of checking that certain constraints are satisfied, and even (in some cases) of cleaning up a large system after it has been put together. Given that programmers are popularly and correctly identified in the public mind as practicing black magic, we have named our proposed junior colleague *EUCRATES*, the sorceror's apprentice of Greek mythology. Unlike that mythological character, however, we want our apprentice to be a diligent, careful helper, who does not overstep the bounds of his capabilites.

We see several past research efforts as having relevance to the tasks we are undertaking, although this project is in many ways breaking into virgin territory. We have already pointed out that we are departing from the approaches summarized in the last section, particularly by virtue of the fact that we see knowledge based programming as the essence of the task. To further sharpen that distinction, let us add that the knowledge which we would wish to encode in our system can not and would not appear as some abstracted or formalized version of the programmer's "common sense" knowledge. Thus, we would not represent knowledge about list structure as theorems in the theory of categories, but rather as the "facts" which every *LISP* programmer knows, namely that there are "cars" and "cdrs", etc. Much of our work will involve codifying enough of these "facts" into the system to get any useful behavior out of it at all.

0.1.2.1 Winograd's "A" System

The type of system we are trying to design was suggested to us by Terry Winograd's "Breaking the Complexity Barrier" paper <Winograd, 1973>. In that paper, it is suggested that a programming environment unifying editors, debuggers, programming language systems and a knowledge base (to be called the "A" system) would be a valuable tool to put at the disposal of the programmer of complex systems. Further, Winograd suggests the use of program annotation to help the system understand the goals, purposes, and methods which the programmer is employing. To this end he identifies three types of comment namely "conditions", "assertions" and "purposes". These will be seen to have their counterparts within our system.

The "A" system as proposed would include: (1) a documentation and question answering facility, i.e. the system could explain various facts about the way the program works as well as insert documentation "on the fly". (2) several levels of interpreters, each capable of a unique tradeoff between efficiency and carefulness in execution. At one extreme, everything is checked and the system runs slowly; at the other, carefully compiled code is run unchecked. (3) An editor integrated into the other sections so that changes to the code can be inserted to fix problems "on the fly", and so that proposed changes to the code can be checked and criticized as they are being made.

What is lacking in this description of the "A" system is any idea of how various parts of the system perform their duties. Our major task will be in filling in these details, which are anything but trivial. The proposed features seem to us to be a fair description of those tools which seem most important, and the work proposed here will center on one of these, namely the question answering - documentation system. However, the ideas which we will employ to accomplish our goals are not touched upon in Winograd's paper; it is not a description of an existing system.

0.1.2.2 Smith and Hewitt's Programming Apprentice

A second piece of related research has been summarized in Smith and Hewitt's "Towards a Programming Apprentice" <Smith and Hewitt, 1974>. Here a system is proposed and described which is intended to achieve many of the aims of the "A" system through a process called "Meta-evaluation". Intended to run within a system based on Hewitt's *ACTORS* formalism, the approach involves certain concepts which we have found very useful. Most important among these is the notion of attaching to every identifiable segment of code a statement describing the behavior of the code; this is intended to say "what" the code does, not "how" it does it. We have used this concept to help us formalize the semantics of program description.

There are, however, extremely important differences between this approach and ours. First among these is that we are attacking a different problem than that addressed by Smith and Hewitt. Their goal is to justify that a module satisfies the contract (i.e. behavioral description) attached to it; to do this they evaluate the behavior of the code on abstract input using an environment of forking contexts and background knowledge. Out of this, they hope to realize a "justification" which captures the teleological structure of the program and to use this to further aid the programmer. Our goal is to build a knowledge base containing such information already and to use it to help the programmer design code. Rather than meta-evaluating code, we try to recognize it as being similar to something which we already understand; we use such recognition to build a model of the code's behavior and teleology.

A second important difference is that we see the structure of the knowledge base as being the essential question, while Hewitt sees the construction of more modular programming styles and formalisms as a central task. Given such an *ACTORS* formalism, Hewitt and Smith believe that

they could build a meta-evaluating system which would serve as the base for their apprentice. Because of this, Smith and Hewitt's system will have to wait for the implementation of a language based on this new formalism, while ours might be of use to the programmer of already existing *LISP* systems. In many ways these two approaches wind up being complementary in the sense that they attack different ends of the same large problem. Each system will be able to incorporate most of the ideas of the other.

0.1.2.3 Sussman's *HACKER*

Many of our ideas about program teleology follow from work reported by Sussman in "A Computational Model of Skill Acquisition" <Sussman, 1973>, which describes a program called "*HACKER*" which can write, debug and learn new programs for the Blocks World. The main ideas we have found relevant center around the notions of "purposes" within a program and the realization of this concept as the functional relationship between segments of code. Sussman identifies two such relationships, namely prerequisite and main step. In addition, he conects these concepts to the temporal sequencing of a program and to the possible causes of "bugs" within a program.

HACKER achieves its ends by attempting to pose a simple solution (i.e. a first order approximation) to the problem with which it is presented. It then runs the proposed program in a "careful" mode in which annotation is checked and a complete history is maintained in the form of process snapshots called the "chrontext". If a violation is detected, the "chrontext" is analyzed and the essense of the goal structure is abstracted from it. This is then checked against a catalogue of known types of "bugs" to find the fix. This information is also used to compile "critics" which will prevent the faulty plan from being proposed again.

Again we find that in many ways our work is complementary to the work reported. In particular, we do not set ourselves the goal of automatic program proposal and debugging, but rather that of interaction with the programmer who is doing these things. Secondly, we wish to have availiable during the design phase, to as great an extent as possible, the kind of knowledge which *HACKER* abstracts at the time of the disaster. Finally, we are working within a domain which is in no sense a toy domain like the blocks world. We, therefore, find ourselves much more often in a situation of partial knowledge, in which interaction with the programmer becomes essential.

0.1.2.4 Goldstein's *MYCROFT*

The final work which has advanced the technology of programming assistance is a program designed to help debug simple programs written by children within the *LOGO* system. These programs are designed to drawn pictures on a display by guiding a "turtle" with simple "forward" and "right" commands. Goldstein's *MYCROFT* <Goldstein, 1974> debugs these programs by comparing the picture actually drawn (actually an internal representation of it) to a "model" of what the program ought to do. Using the model and the code, the system discerns what the "plan" of the program must have been and from this generates the program's annotation. This is then used to guide the debugger in finding the problem and proposing a correction.

The differences between this approach and curs are mainly those already stated, namely that we wish to aid in the design of programs which are more complex than those possible within the limitations set by Mycroft and that we see debugging as being only a part of that process. Moreover, to accomplish our aims, we think that the user must specify his "plan" in advance, and that, in our domain, it would be extremely difficult to figure out the "plan" without having hints and a similar "plan" within the knowledge base. We do find, however, that Goldstein's notion of

using the plan as a driving force in debugging is an extremely valuable contribution. Furthermore, his classification of plan types has provided a starting point for our thinking.

0.1.2.5 Greg Ruth's Sort-Program Debugger

One other thesis done recently at M.I.T. bears some relevance to our overall goal of building a system capable of analyzing and understanding programs. Greg Ruth <Ruth,1973> constructed a system which is capable of debugging sorting programs written by children in an elementary programming class. Ruth's system knows several different sorting algorithms (e.g. bubble sort, interchange, etc.) and uses these as the driving force of the debugging session. Bugs are found by first finding that algorithm which most closely matches the student's program, and then classifying all differences as bugs.

Although there are superficial similarities between this approach and that which we will present here, the essense of the two systems are essentially dissimilar. Like us, Ruth wants the driving element of his system to represent a class of programs. He therefore represents his algorithms as production rules in a context free grammar. Recognition, or more appropriately matching, can then be handled by a simple parser which takes the students program as text and parses it against the algorithm grammar. This procedure, it seems to us, inherently limits the system to working within a remarkably narrow range of permissable programs and, therefore, would seem to be an unlikely candidate for further development. Furthermore, the approach seems incapable of providing much assistance to a sophisticated programmer during the design phase.

0.1.2.6 Others

The works cited above have been mainly useful to us in clarifying *what* kind of knowledge the programming assistant would have to be in control of. None of them address the issue of representation of knowledge within a large domain. (To estimate the size of our domain, we can start with the fact that *MACLISP* has over 100 subrs available to the user, and that the basic techniques which the average programmer calls upon might well be an order of magnitude greater in size. Then, there are more involved concepts, such as those summarized in Knuth's several volumes). Our thinking on this issue is still largely unsettled but to the extent that we have ideas they have been influenced by Minsky's "Frame Systems" paper <Minsky, 1974> in which the idea of "chunking" the knowledge into "frames" with "slots" and "default values" was presented. Much of the structure we will present here has this flavor to it. However, the other main thesis of this paper, namely the "hypothesize and jump" paradigm of recognition represents an idea which we have yet to explore very deeply. In some regards, we find the paradigm of recognition presented by Marcus in his "wait-and-see" parser <Marcus, 1974> equally compelling. It is in this area of representations and its relationship to understanding and recognition paradigms that our ideas are in the greatest need of clarification.

0.2 SCENARIOS

In this section we illustrate some of the behaviors which the programming apprentice should have. Most of these will be seen to fall into the category of design and coding advise, i.e. they help the programmer avoid errors, or they catch the errors before they get entangled into a complicated web of the design. Most of the examples presented here are real in the sense that the knowledge which the apprentice calls upon in these examples was also used by the author in writing code very similar to that presented here. Most of the mistakes shown here were real bugs in the code. It is also interesting that this code is part of the P.A. system; it is part of the mechanism already created to attach annotation to code in such a way that both are accessible during program execution, editing, etc.

In this presentation all dialogue is in English. This is primarily for reading ease. Although we feel that the system we are designing will have adequate knowledge to communicate in natural language, we also feel that the essense of our research ought not to be in that direction at the current time. In the dialogues which follow the programmer will be indicated by the Roman text, while the Apprentice will be indicated by the Gothic. Program code in UPPER CASE will be the output of the apprentice, programs in lower case are to be understood as the programmer's input. The *ttalics* text are explanatory comments added by the authors to help indicate what the apprentice is doing.

0.2.1. First Scenario: Initial Design

I'd like to build a hash table

O.K. you'll need an insert, a lookup, an array, a hasher, and optionally a delete routine.

The P.A. knows the main parts of a hashing system.

Here's the code for the hasher.

```
(setq htable-size 8179)
  (array hash-table t htable-size)
  (defun hasher (key)
    (hash-table (abs (\ (maknum key) htable-size))))
```

O.K. let's do the insert routine.

I'd advise using lists for buckets

The P.A. knows that there are several ways of doing hashing systems. It also knows that in LISP the most common approach is to use lists.

That seems alright; what are other choices?

Rehash, linear and overflow.

P.A. gives name for alternative schemes. It could explain any of these but the programmer does not ask him to.

O.K. let's use lists. Give me an outline.

Programmer asks for some hints on how to structure the lookup routine.

The lookup takes one argument, the key. It has 2 steps:

1. hash to get a bucket.
2. lookup the key in the bucket.

The p.a. knows the structure of this type of lookup and gives a broad outline of it. If asked to it could go into much greater detail.

O.K. here's my code.

```
 (defun lookup (key)
    (do ((*f (hash key) (cdr *f)))
        ((null *f))
        (cond ((eq (caar *f) key) (return (car *f))))))
```

Is it correct that the car of each entry is the key?

The P.A. figures out from its knowledge of the program's structure that the first element of each entry must be the key.

Yes, Now let's do the insert.

I'm sorry, but there's a mistake on the second line. By (hash key) you must mean to use the hash routine, but its name is hasher not hash. Should I fix that?

The P.A. catches a "spelling" mistake. It does this by seeing that a particular function application must be the call to the hashing function. Since it already knows that this routine is named hasher not hash, it both spots and fixes the error. It is important to realize that this check would have worked even if the correct name and the name typed bore no similarity to each other. The check employed here is semantic, not a syntactic check for close to correct spellings of common words as is done in INTERLISP.

Yes, then give me an outline of the insert routine.

Insert takes a key and the data item(s) as the arguments. In general it has two steps:

1. Hash to the bucket
2. Insert the entry which is built from the key and the data item(s) into the bucket.

Again the P.A. uses its knowledge of the structure of hashing schemes to advise the user on how to proceed.

O.K., here's the code:

```
(defun insert (key parent name)
  (store (hasher key) (cons (cons parent name) key)))
```

I'm sorry but I see a few problems:
1. You're building an entry here by doing:

(CONS (CONS PARENT NAME) KEY)

But in the lookup routine the car of the entry is the key.
2. The first arg. to store must be a subscripted array reference, but hasher is not an array. It returns one member of the array hash-table, namely a bucket.

The P.A. knows about the possible dependencies in the hashing scheme, in particular, that the entry builder of insert must agree in certain details with the key-extractor of the lookup routine. Also it knows syntactic restrictions of the basic LISP functions.

Let's fix the first. Reverse the order of the arguments to the outer cons.

It is worth noticing that, although the P.A. notices the bug, it is the programmer who proposes a solution. This will be the normal mode of operation.

I assume you mean the following, it will fix the first problem.
(CONS KEY (CONS PARENT NAME))

The P.A. rechecks the problem and sees that it is now fixed

Let me see the second problem again.

In the following form, the first argument must be a subscripted array reference:
(STORE (HASHER KEY) (CONS KEY (CONS PARENT NAME)))

Notice that the first fix has been carried forward, i.e. the patch is now in place

What is the name of the array?

hash-table

change the first arg. to (hash-table (hasher key))

I assume you mean:
(STORE (HASH-TABLE (HASHER KEY))...)
That won't work either. Hash-table is an array; its arguments must be numbers. But hasher returns a bucket which is a list.

The P.A. again checks for arguments being of the right type. This time it can do this because it knows the definitions of hash-table and hasher. It checks to see if the one produces the right kind of data for the other. Since they don't it complains.

I'm going to change hasher so that it will return the subscript of the bucket rather than the bucket?

You can't. In the lookup routine hasher is required to return a list so that it can be searched by cdr'ing down.

The P.A. here searches through its knowledge of the program structure to see what might depend on the current definition of hasher, and if the change will break that dependency. It sees that there would be a problem in this case.

O.K., let me see the code for hasher.

The programmer is intending to use the old code as a model for writing a new module.

Here it is.
```
(DEFUN HASHER (KEY)
    (HASH-TABLE (ABS (\ (MAKNUM KEY) HTABLE-SIZE))))
```

Rename this one bucket-hasher. Also make a function called number-hasher out of the inner form. It will take a key as its argument and return the subscript in the array of the appropriate bucket.

Is this what you want?

```
(DEFUN NUMBER-HASHER (KEY)
   (ABS (\ (MAKNUM KEY) HTABLE-SIZE)))

(DEFUN BUCKET-HASHER (KEY)
     (HASH-TABLE (ABS (\ (MAKNUM KEY) HTABLE-SIZE))))
```

The P.A. performs a simple editing function, but also uses its knowledge of the program's structure to build itself an internal model of the new function, so that it can answer questions about it also.

Yes. Now change the inner form in bucket hasher to call number-hasher with key as the argument.

```
(DEFUN BUCKET-HASHER (KEY)
     (HASH-TABLE (NUMBER-HASHER KEY)))
```

The P.A. uses its knowledge of the new function to see that the call to it is legitimate, and that the old function continues to behave as before. Since this is true, it now notices that the name change must be propogated to everything that depended on this routine.

Should I change all references to HASHER to BUCKET-HASHER?

Yes, except for the one in INSERT. Change that to NUMBER-HASHER. Then check out the insert routine.

```
(DEFUN INSERT (KEY PARENT NAME)
  (STORE (HASHER KEY) (CONS KEY (CONS PARENT NAME))))
```

How do you want me to fix this?

Change the first arg. of store to

```
(hash-table (number-hasher key))
```

```
(STORE (HASH-TABLE (NUMBER-HASHER KEY))
          (CONS KEY (CONS PARENT NAME)))
```

O.K. but there is still another bug. After you do this store, there will only be one entry in the bucket. If there was anything in the bucket before the store, it won't be there afterwards.

The P.A. notices that the insert routine will not satisfy its description of an insert, namely that it adds a new element without deleting old ones. It does this by considering the effect of each of the steps and then trying to demonstrate that the overall specifications are met.

Let me see the form starting with the cons.

```
(CONS KEY (CONS PARENT NAME))
```

o.k., change that to:

```
(cons (cons key (cons parent name))
      (bucket-hasher key)).
```

You now have a working insert routine:

```
(DEFUN INSERT (KEY PARENT NAME)
  (STORE (HASHER KEY)
    (CONS
      (CONS (CONS PARENT NAME) KEY)))
      (BUCKET-HASHER KEY))))
```

The session continues until both the user and the P.A. are sattsfted. The program, together with all the knowledge about it are ftled away for future use.

0.2.2 Scenario 2: Working on An Existing Program

I'd like to work on the hashing system again.

O.k.

What does an entry look like?

```
(KEY. (PARENT.NAME))
```

The P.A. displays a "typical" entry of the hash-table. Notice that it constructed this during the session presented in the last scenario by applying its understanding of the structure of the code to identify the entry composing segment. From this it constructed a "typical" entry.

What are the restrictions on key?

None.

Call entry-1 any entry whose key is an atom.

```
entry-1 is-a entry
(key entry-1) is-a atom.
```

The apprentice reports that it has added a new model to its knowledge base.

When is ENTRY referenced?

In the lookup routine, where its key part is checked, while looking up the entry in the bucket. In the insert routine where the entry is built and inserted into the bucket.

The apprentice checks through its model of the program to see where the conceptual entity ENTRY is referenced and how. Notice that there is no modularized routine to build an entry or to extract the key from an entry. The apprentice's model of the program, however, does contains such concepts with pointers into that part of the code which realizes the behavior of these constructs.

I am going to make a change to the lookup routine. If the entry is an ENTRY-1, then the test in the bucket lookup will check to see if the parent part of the entry is equal to an additional input called *parent.

The programmer gives the apprentice information describing modifications to the plan of the program. The apprentice will try to fit these mods into its existing models of the program structure so that it can check that the programmer's changes to the code will accomplish what he

intended to.

Since the only distinguishing feature of the ENTRY-1 is that its key is an atom, I will assume that the check for ENTRY-1 is a check that the key is an atom.

The apprentice explains the conceptual changes it has made. Notice that it is working with the concepts KEY, ENTRY, etc. rather than with car's etc. Then if the programmer changes the format of the entry, the model is still correct at the top level.

The programmer continues working on further modifications. However, we feel that the general capabilities have been indicated sufficiently.

0.3 THE GAME PLAN

0.3.1 Outline of the Proposal

The remainder of this document will present a summary of the framework which we believe will be a sufficient base for the capabilities of the apprentice. We believe that the essense of our task will be to create a knowledge base which captures the semantics of programs. Chapter 1 will present our preliminary thoughts on this matter. Chapter 2 will further develop these ideas and show that the framework developed is adequate to account for most annotation which we have seen attached to programs. This is of extreme importance, for we believe that annotation is an extremely valuable and often vital aid in program understanding. Chapter 3 will then explain how the structures developed in the first two chapters will allow the system to understand a piece of code it has never seen. By understanding, we will mean the creation of a model of the program sufficient to answer questions and to identify bugs. Because this process will bear such similarity to classic AI recognition problems such as vision and natural language understanding, we will most frequently refer to it as the *recogntttion problem.*

0.3.2 Research Plan and Schedule

It is important to realize that, at the current time, virtually no code has been written. The research we are proposing can, therefore, best be explained as creating the programs which this document hints at. Our basic and firm belief, reflected in the organization of this document, is that the same foundation underlies all the various tasks we would wish the P.A. to perform. This foundation is the knowledge base and the modelling of program teleology developed in Chapters One and Two. Of the many roles the P.A. can play in assisting the programmer, we will choose one to be a demonstration of the viability of our ideas--*program explanation.* Namely, we will build a system

which will be capable of answering all the "wh-" (what, why, when, where, how, etc.) type questions about an arbitrary program using hash-tables. Further, it should have the capability to answer questions about dependencies within the program. Finally, it should have the ability to detect (but not correct) bugs. We feel this is a good choice, because it will make it as explicit and convincing as possible that the P.A. really "understands" programs. We will probably limit the LISP code we will handle to several basic functions such as: cons, car, cdr, rplaca, rplacd, prog, progn, do, go, return, cond, and, or.

In order to accomplish this we see two main tasks immediately ahead. The first is the data base design. Although we feel that we know what needs to be in the knowledge base, the question to be immediately settled is how is the knowledge base to be structured. We plan to settle this within the next month. Secondly, the recognition task must be studied further. Given a fixed design for the knowledge base, we feel that the next two months might well be spent in exploring this area both in terms of limiting the problem to a manageable size and then in terms of actually writing some code. Having done this the remainder of our time will be spent on examining how to make the question answering-bug detection system work.

0.3.3 Towards A Programming Apprentice

This section will simply be an outline of those capabilities which we feel a complete programming apprentice ought to have. When our research is completed, we will be able to present a detailed account of how we implemented a program explainer using our knowledge base and modelling techniques as foundation. Furthermore, we will also give in the final research report hopefully convincing presentations of how the same foundation could be used to implement the remainder of the following behaviors:

(1) Program Explanation

-answering "wh-questions", why, what, how, when.
-explaining behavioural relationships between code segments behavioural relations between segments of code
-generating summaries of program structure

(2) Debugging Assistance

-ideas of Sussman, Goldstein, Hewitt
-closely related to informal verification

(3) Automatic Coding

-on a local basis, as different from "automatic programming"
-also possibility of "cleaning" up code, i.e. rewriting it making surface structure reflect underlying model more clearly
-automatically generate extra annotation from information in knowledge base

(4) Intelligent Editor

-check for propogation effects of changes
-user can give editing instructions in semantic rather than syntactic terms

0.3.4 Resource Requirements

In order to accomplish the tasks which we have set for ourselves, we will need a large amount of computer time. Fortunately, that time is available to us on the A.I. Lab's PDP-10. Other than this, we have no requirements for resources or materials. We are, therefore, prepared to carry on our research without further resource allocations from either the department or the Institute.

0.3.5 Division of Effort

This work is being conducted as a joint project, precisely because its structure defies natural divison. In particular, the knowledge base is the key to the whole system. If it is properly designed, then the intended application parts of the system (i.e. the recognition system, the question

answering system and the bug detection system) will be relatively simple to implement. We, therefore, find it virtually impossible and certainly inappropriate to state a division of responsibilty. Both of us are accountable for the whole project; it is both of our responsibilites as well as that of our advisors to guarantee that the work is, in fact, shared equally. We believe that the rest of this document will indicate that this method does indeed work; the work presented here has been work done mutually. In fact, it has been an exciting phenomenon so far that we are acting as each other's advisors more than our official advisors are (although they are by no means shirking their duties).

CHAPTER ONE. THE KNOWLEDGE BASE

1.0 Introduction

In the preceeding scenarios, we presented several examples of useful and desirable behavior which we might want a programmers helper to be able to perform. An essential aspect of each of these is, in our view, that the programmers helper would have to be knowledgeable, i.e. capable with only a little help of "understanding" what we are doing. We feel that the essense of such understanding is the existence of a large base of active knowledge containing substantial information from the domain of programming which is structured in such a way that the relevant information can be called into use in the apropriate situations. Given that the knowledge we are referring to is familiar to any programmer, our goal will be to design the appropriate structure and then to load into the data base a representation of some small segment of programming knowldege, so that the programming apprentice can perform its services.

In general, the goals of our research will be centered around this approach of understanding a program in terms of already existing knowledge; we feel that this dictates that a major part of the overall system must be dedicated to the task of recognizing the conceptual structure of a program and to the identification of those concepts which are used in the course of a program. Frequently, this is anything but trivial. For example, hash tables, which we will use as a running example, consist of several functions (i.e. insert, delete, etc.) and several data structures (lists, arrays, etc.). It is the totality of the code used to represent all of these which, in fact, constitutes a hash table. In addition, as anyone familiar with programming knows, the actual code to make a hash table will vary tremendously from one programmer to another. Nevertheless, the understander of such code typically understands it by constructing a model of the program's behavior which depends very

little on the particular hackery of the coder. In fact, the kinds of description used in such a model, and their arrangement seem to be remarkably more predictable than the actual code.

It therefore follows that the knowledge base, which is to be the central tool of this understanding process, should be structured in such a way that it can be viewed as having semi-discrete units which the programming apprentice can use as conceptual building blocks in constructing a model of the program. Given such building blocks to refer to, the programmer can tell his apprentice that a particular segment of code corresponds (more or less) to one of these conceptual building blocks, thereby, associating with this segment of code all of the knowledge contained in the referenced conceptual building block. This will allow the programmer to comment his code largely by comments of the form "this is a foobar" or "using the frobbie technique", rather than by having to include detailed descriptions or explanations of the program's goal structure, etc. The latter type of detailed annotation is in practice difficult to formulate in a line by line format and is therefore usually completely avoided by professional programmers.

The building blocks, out of which the knowledge base is constructed are, therefore, to be regarded as generalizations of programs, rather than as representations of a specific segment of code. For example, the node in the knowledge base corresponding to "hash table" will be a single conceptual unit which can instantiate itself to any of the various implementations of a hash table, while yet maintaining that knowledge which is true of hash tables in general. Since objects in the world of programming are characterized sometimes by what they do, sometimes by what they are good for, and other times by how they are internally structured, all of this information will be present in the building blocks. The rest of this chapter we will show that this knowledge will fall into three categories, namely: *descriptive models*, *plans*, and LISP specialized knowledge.

1.0.1 Design Criteria, A Priori

Our current ideas about the structure of the knowledge base are formed and motivated from two directions. First, we have to establish what knowledge is in a fundamental sense *sufficient* for the P.A. to be able to perform the kinds of tasks we have in mind. One way we get a feeling for this is to imagine specific performance scenarios, and then satisfy ourselves that, programming and implementation issues aside, the knowledge required can be accounted for and is somehow present or at least implicit in the system. An example of applying this methodology is to consider the answering of "WH-questions", i.e. what would we expect the P.A. (in explanation mode) to answer when the user points to a segment of code and asks a question that boils down to a form of How?, Why?, Where?, What?, When?, etc. First we have to decide what the programmer would have in mind when asking such a question, and then assign one or more internal operational definitions which will correspond to the programmers meanings, but may then be implemented technically. Such an exercise may define a fundamental capability or basic class of required knowledge, e.g. respectively the ability to cross-reference and index the code and generate unambiguous explanatory references to locations in the code in answer to "where" questions; or, in answer to "when" questions, we must develop a time representation appropriate to the expression of timing relationships in the domain of programming. Alternatively, as in the case of "how" and "why" questions , a performance scenario may give us insight into how particular aspects of the knowledge base would actually be used. Thus, one major input into the design of the knowledge is motivation from the intended applications.

The second important design criterion is that we are striving towards a *naturalistic* representation, i.e. that the organization and interrelation of concepts in the knowledge base parallel as closely as possible the way human users naturally conceive of their work and express themselves. Thus any

terminology or grouping of concepts that are natural to human programmers should be reflected in the layout of the knowledge base. This implies, of course, quite a bit of redundancy in the knowledge base. We are not primarily interested, for the purpose of constructing a P.A., in finding "minimal" abstract formalisms for the concept space. Keeping the organization of the knowledge base naturalistic will also facilitate the development of the system in the direction of understanding user comments and generating explanations in natural language.

The major character of the knowledge base emerging from the two criteria above is that it necessarily contains a variety of representations. In the following subsections we will describe some of these that we have ideas about at this time. Though it is not always easy, we will try to separate questions of implementation from more basic questions of representation.

1.0.2 Justification of Hash Tables as Research Example

In our research so far we have used the example of a program using a hash table and associated programming concepts as an aid to guiding and stimulating our thinking about the problems of building a P.A. The issues discussed in this paper will also be illustrated primarily by examples drawn from this subdomain of programming techniques. The fact that we have found this example useful is certainly a most important, and possibly totally adequate justification, but we would also like to stop a moment and justify this choice on a more theoretical, though admittedly somewhat post hoc basis. Let us try to get a feeling for how large the conceptual space of programming techniques might be, and then how much of this space is covered by our chosen example of hash table programs. The space might be divided into two areas, program dynamics or control structure, and program data structures. Under each heading we might list loosely all the forms we can think of. Such a list might be:

Data Structures:	arrays	stacks
	lists	sets
	trees	bags
	tables	rings
	property lists	queues
Control Structures:	iteration	recursion
	dispatching	backtracking
	subroutines	pattern directed invocation
	coroutines	parallel processing
	interrupts	

Of these topics, hash table programs introduce the following subsets:

Data Structures: arrays, lists, tables, rings

Control Structures: iteration, linear plans, subroutines, recursion

Our justification is then that these are a reasonable number of basic concepts to expect to be covered by one example. In choosing additional research examples, we will try to cover complimentary aspects of the domain.

1.1 THE CONTENTS OF THE KNOWLEDGE BASE

The knowledge base is required to contain an adequate representation for understanding programs. We distinguish three broad categories of such knowledge. The first, *descriptive models* are intended to answer "what" type questions. The second type of information, which we call *plans*, is explanations of how various behaviors are realized. Finally, information about the semantics and typical forms of LISP code must also be contained in the knowledge base. The rest of this chapter will explore these domains in some detail.

1.1.1 Descriptive Models

1.1.1.1 Conceptual Relatedness

There seems to be a deep-seated dualism between object and process in the way people talk about the entities in the domain of the P.A. For example, a hash table can be thought of either as a concrete object consisting of an array of association lists, or as an entity whose behaviour is described by the laws of associative retrieval. In fact, neither of these is a complete description alone. There are, after all, several techniques of associative retrieval, and arrays are used for many things besides hashing data. Probably both kinds of description would be expected in answer to the question "what" is a hash table. Thus in our knowledge base we will need to be able to capture both flavors of description. The first kind is what might be called *conceptual relatedness* information, and leads us to think of implementations like Winston Nets with relational pointers telling what is part-of or a-kind-of something else, what depend-on something else, etc. This provides the decomposition sense of the answer to "what".

1.1.1.2 Intrinsic Descriptions

Also, attached to some concept (or node in the net) we see a need for a behavioural description. In this regard we are prone to follow Carl Hewitt's notions and speak of specifying the behaviour in terms of its incoming expectations or *preconditions* , and its outgoing entailments, or *postconditions.* These together constitute the *intrinsic* description, or what we will call *specs.* The intrinsic description of a hash table deletion routine for example, would contain clauses which, notation aside for the moment, would express the following.

```
Intrinsic Description (Specs) for HASH-TABLE-DELETE

        Precondition:  Well-formedness of input arguments .
        Postcondition:  The item to be deleted is not in the table.
```

In most cases, the object which is represented by a descriptive model will have a range of behaviors, such as insert, lookup, delete for the case of hash tables. In terms of the code, this might well be represented by there being several functions which, taken as a cluster, comprise this total repetoire; in fact, most decent LISP programmers would use such a implementation. Moreover, conceptually these various capabilities represent a unified whole. We are, therefore, led to seeing the intrinsic description as a collection of cases which collectively describe the objects total range of behaviour under all conditions.

An important point that will develop from our hash table example is that we often find it natural to describe the behaviour of entities partly in terms of their interaction with other entities. In this case there are important interactions between the insertion, deletion, and lookup components of a hashing scheme, for example that if you insert an entry and then delete it, a subsequent lookup will fail. These kinds of relationships between segments of code are properly part of their intrinsic descriptions, because they are independent of the surrounding teleology, or goal structure in which

the components are employed.

However, a segment of code may also have *extrinsic* relationships to other entities. For example to say that a hash table represents the currently reserved airline seats, is to give an extrinsic description of the hash table. At a different level, the information that the programmer's *purpose* in calling a particular subroutine is because one of its postconditions is the precondition of a subsequent segment of code, is another kind of extrinsic description. The purpose of a segment of code is part of its extrinsic description, and will often vary if the code is used in several different places. However, the intrinsic description is always the same. The arranging of code into an interwoven structure of compatible purposes is very much the essence of the programmer's occupation. The basic schemas that he uses in arranging this teleology is what we will refer to (following Sussman) as *plans*. These will be discussed in more detail in a following section.

1.1.1.3 Deductive Reasoning

On a primarily introspective basis we feel (in disagreement with Winograd in the "A" Paper) that sophisticated deductive capabilities are not the major bottleneck in constructing a P.A. It is our observation that in reasoning about their programs in the contexts of debugging or informal verification, people typically employ only rather short direct lines of deduction. People do not naturally verify their programs in the strict sense of Floyd, for example, wherein the efficiency of an automatic theorm prover for the first order quantificational logic would be a major issue. Rather, it seems programmers use a "common sense" mode of reasoning, wherein the knowledge is looser and quite wide-ranging. The key problem is to choose the relevant information, and once this is done, the deductive steps are usually few. For example, suppose the P.A. was faced with a hashing program in which at some point an item that was expected to be in the table, failed to be

found by the lookup routine. The logical place to look for clues would be in the *specs* of the various routines participating in this process, especially because the precondition-postcondition pairs have a strong deductive flavor to them. (i.e. given that the precondition is met, and that the routine was called, it is valid to assert the postconditions). The relevant information in this case might be represented in some simple logical notation, of the following flavor:

```
((Insert (entry key data) -> (member (entry key data)))

((Delete key) -> (erase '(member (entry key ?))))

((Member (entry key data)) -> ((Lookup key) = data))
```

Notice that this is not intended to imply that the predicate calculus would be a good representational scheme for this kind of knowledge. Quite the contrary, the need to use an operation like Erase (in order to represent side effects), as well as a need to keep track of which facts depend on what other facts , e.g. (lookup key)=data) should not remain true after (member entry key data) has been deleted, clearly suggests that a procedural, data-base language is advised for doing logical deductions. However, we also do not believe it is the case that gulping up Micro-Planner or Conniver whole hog will meet the needs of the total system we are constructing, either. We do have some ideas at this time on how we might construct our system to have the desired properties. The essense of the idea is localization of the reasoning process, so a simple deductive mechanism will not be swamped by irrelevant theorems.

The deductive facts described above provide one example of this localization, in that they will be attached to the appropriate descriptive models, rather than hanging loose in a CONNIVER or MICRO-PLANNER data base. The reasoning component of the P.A. might then have several (7 plus or minus 2?) "scratch-pad" deductive databases (a la Micro-Planner). When a particular concept becomes involved in the current focus of reasoning, its associated theorems (e.g. the

program specs, used for deductive purposes) are brought into the current deductive scratch-pad. If a conclusion can still not be reached, the P.A. might consider widening the focus, thus bringing in more, but potentially less relevant, information. Reasoning may also take place simultaneously at several levels of abstraction, so there must also be a mechanism for communication between the databases.

1.1.1.4 Examples

We have taken seriously the common observation that one of the best things about the new MACLISP manual is its generous use of examples complimentary to the definitional explanations. Clearly, any system that claims to be at all anthropomorphic in its behaviour, must have the ability to manipulate and reason with examples. It turns out upon reflection that examples are often a compact way to implicitly represent knowledge about the behaviour of an entity. Therefore, we find it useful to allow the descriptive model to contain "typical" examples in addition to the other information already described.

For example, in the abstract, if you have an object X to which can be applied operations A - Z with varying results, explicitly you would have to represent this information something like:

```
(A X) = R1

(B X) = R2

(C X) = R3 etc.
```

But if you have the system interpreter easily available (e.g. a "careful" version of the LISP interpreter), you can simply make a temporary copy of the object X, apply the operator of interest in a scratch-pad context and "see what happens".

Examples are also often a convenient way to teach (i.e. input) new concepts. For example, to explain a new data structure FOO, rather than giving a list of constraints, the user might find it more convenient to give a canonical example, e.g.

```
( ( ( X Y Z ) . A ) . B )
```

Then if later an execution interrupt occurs when trying to take the CDDAR of a FOO data object, the P.A. might hypothesize that the bug is due to the ill-formedness of the data object (rather than due to incorrect processing). Evidence for this hypothesis would then be obtained (and this is the way people operate) by attempting to take the CDDAR of a known example of FOO. In this case it would be found that CDDAR is illegal, suggesting that something is wrong with the process that is requesting the CDDAR to be done, rather than with the process that formed the FOO.

There is of course a large area of research in determining what exactly constitutes what is passingly referred to above as a "canonical example". It is in a sense true that people in such situations are abstracting a higher level description from the example presented. In fact there are certain "culturally" accepted conventions and heuristics that are used to help this process of understanding examples. For example, if you see the list

```
' (MARY HAD A LITTLE LAMB)
```

you usually interpret this as meaning an arbitrary string. Obviously, we are not seriously suggested that a P.A. needs to have a comprehensive knowledge of fairy tales. Rather, we simply wish to point out the existence of certain informal notations (e.g. less formal than a strict pattern syntax). Another common conventional interpretation is that higher multiplicities (greater than 3 or 4, say) usually indicate the generalized "n-multiplicity" case. For example, a function with a variable number of arguments is not usually illustrated with only one or two arguments, since many people might find such an example misleading.

It would also be desirable, but a somewhat harder problem, for the P.A. to be able to generate its own examples from more abstract descriptions. This capability could become a very important and powerful aspect of its reasoning apparatus. Finally, and this is a complete research project in its own right which will certainly not be addressed here, it would of course be useful if the P.A. could conversely generate abstract descriptions from one or more examples.

1.1.1.5 Typical Bugs

There will be a whole class of information in the knowledge base concerned with "bugs". How the P.A. would use this information to assist the programmer in debugging was hinted at in the scenarios of Chapter Zero. Let us just mention here for completeness in the description of the knowledge base that attached to various nodes would be information about typical bugs that are associated with them. Frequently, such information might already be implicitly present. For example, in the descriptive model of an array, the specs require as a prerequisite that the args must be "in bounds". The function of the additional information on bugs is to advise the P.A about what bugs are likely to appear. This information is of heuristic value in debugging sessions.

1.1.2 Implementation Plans

Another form of information which clearly must be kept in the knowledge base might well be though of as implementation plans. It is important to realize that given an object, there are typically several ways of achieving the desired behavior. In our example of the hash table there are in fact three rather well established implementation plans. One can use the hash-rehash scheme, overflow tables, or lists to implement the required behavior of a bucket. In numerical calculations, square roots might use the successive approximation plans (Newton's method, the halving method) or alternatively a series expansion might be employed. Virtually all interesting

computations have these varieties of implementation plans available to them.

The choice of implementation plan is a choice which is typically made once and thereby sets the context for much further understanding of the program. As an example, consider the implementation of a queue. Virtually every time-sharing system maintains several queues, and they are frequently implementaed in different ways. One typical method is to use a LISP style list. The characteristics of this method is that free storage is somehow linked together, and that entries on the queue are chained together by forward pointers. Typically, such queues are used where entries are either entered at places other than the rear (say threaded in by priority), or where entries can be removed (for example, a quit or phone disconnect forces the entry to be removed from the allocation queue).

On the other hand, it is frequently the case that such behavior is not needed and that a simpler method can be employed. Namely an array can be used with a front and a back pointer. Removal and insertion of items is accomplished by moving these two pointers. Garbage collection is not needed.

Given that this choice has been made, it is clear that a context has been set for understanding the program which implements the plan. If a LISP style list structure is employed, references to the forward and backward pointer, or to the array are unlikely to make much sense. Similarly, if one had in mind an array oriented queue with front and back pointers, then a reference to the free storage list would be out of context. Even worse, there might be cases where the same concept served different functions in two different implementation plans.

Clearly the descriptive model (in the sense we used it in the previous section) must point at all of the implementation plans, yet once a choice of these various plans has been made, it is as if the others were blocked out. Concepts, objects, specifications, etc. are only relevant to that implementation plan which is active.

The clearest distinction we can make between implementation plans and descriptive models is that the latter explains *what* an object is, while the former explains how that behavior is to be realized. In particular,the plan is intended to present a *high level, goal ortented* description of how the behavior is to accomplished. Thus, a minimal plan would be just a sequence of what other segments of code are to be called upon. Such descriptions are, however, are by themselves misleading. Consider, for example, the "plan" to build two 3-block high towers. Simply enumerating the steps we would get the following plan:

1. Put b on a
2. Put c on b
3. Put e on d
4. Put f on e

Although this is a correct procedure, it is misleading as a general plan for accomplishing the stated goal, because steps 3 and 4 clearly do not have to come after steps 1 and 2.

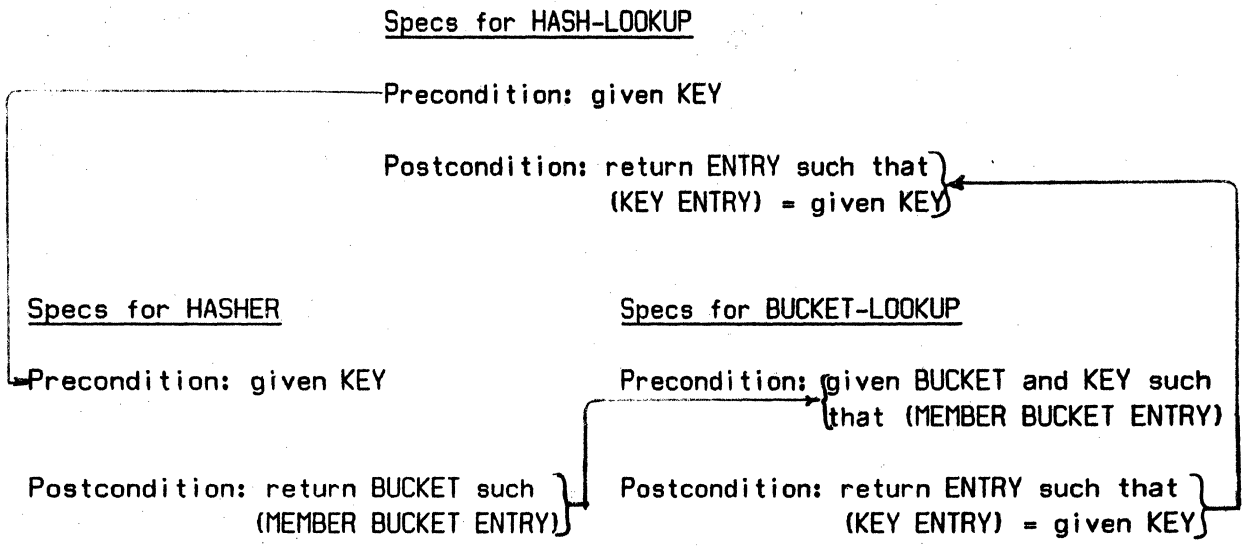
In contrast, consider a "plan" to find a hash table item. Our current approach would have us state the plan as

1. Hash to the bucket
2. Lookup the item in the bucket

Clearly this is not misleading as the previous example was, since this ordering is required. The

reverse ordering of doing a random bucket search followed by a hash would produce garbage at best. Thus, this aproach of "listing the goal steps" can be seen to be inadequate to explain the full richness of the intertwining of the various steps involved in the plan. This inadequacy occurs precisely because the simple notion of ordering of "high level" steps has no notion of *purpose* within it. That is, a more complete notion of plan would require that we specify not only *what* the steps are (in a high level, goal oriented fashion), but also *why* each step is thought to be valid and *how* it helps to accomplish the overall goal.

The key to giving a clear semantics to such a notion is in realizing that any segment of code which we are talking about has (at least potentially) a descriptive model containg the *specs* of that segment. These specify what the code can do. A *purpose* will be defined as a correspondence of the postconditions of one set of specs to either the preconditions or postconditions of some other set of specs (perhaps even its own). For example, in a hash table lookup, the postconditions of the specs for the hash step are that a bucket is returned which contains the required entry. This corresponds to the precondition of the bucket lookup routine, which requires that the entry be present in the bucket, and promises as a postcondition that it will return an entry with the required key. This in turn, corresponds to the postcondition of the hash table lookup routine. Thus the following scheme exists:



In general, plans do not tend to be this simple. For one thing, frequently there will be several purpose arrows eminating from one set of specs, indicating that several courses of action will be pursued at this point, i.e. that there are independent sub-goals which can be pursued in any order. Secondly, even in this plan there have been simplifications made to ease the exposition, one of which is to only present the top level of the plans, i.e. no indication is here given of how hasher or bucket-lookup achieve their specs. This is as it should be, since that information is clearly in the plans for these sub-steps which can be found by going to the descriptive models of hasher and bucket-lookup and asking for their plans.

Although we will go into this in greater detail in a later section, we will point out here that the plan imposes limits on the ordering of the steps in the actually realized code. Clearly, if there is a purpose link between *A* and *B*, then *A* must precede *B* in the actual execution of the code. (In the case of a loop it will be true that both *A* has a purpose link to *B* and the reverse. I.e. both must precede each other, hence a loop). On the other hand, if *A* has purpose links to *B* and to *C* we can only say that *A* must precede both of the other segments; the ordering of *B* and *C* is

undetermined. Therefore, a plan imposes a partial ordering of step execution. If this ordering is further specified into a strict total ordering, then we will have produced a *flowchart* of the computation to be performed.

In summary, then, the plan provides both the explanation of how a computation is to be accomplished and a generalized ordering of the steps which can be instantiated into a flowchart of an actual segment of code. In general, the answer to a "how" question is contained in the plan. On the other hand, "why" questions are typically asked about a particular segment in the context of a particular plan. Answers to such questions are contained in the information provided by the purpose links of the plan. For example, the answer to "why is the bucket lookup called" in the above example is that the bucket lookup will find the desired item if it's in the bucket, and the hasher routine guarantees that it will be in the bucket if it's in the hash table at all. More succintly, we could have said that it was called to return the item. In any case, the information was in the plan. Thus, almost all of the information we need in contained in the plans and the descriptive models.

1.1.2.1 Representation of Time

One of the WH-questions which the P.A. will of course be called upon to answer is "when". For example, "when is the variable x bound?". The most basic way to answer this in the context of programming sequential machines is in terms of before and after, e.g. "x is bound after y is set to NIL, and before F is called." Thus, our model of time at this level is simply the *flowchart* of the program. This, in turn, as we showed in the last section, is nothing more than an instance of the implementation plan for the program. The important and difficult issue here is to determine which reference points will be relevant to the programmer's current intentions, and will thus

constitute the "correct" answer to his question. Programs also have the notions of duration and contemporaneousness, both in the sense of coextensive intervals and coincidence at a point in time. For example, "During the execution of the interpretation functions, the value of PTR is the current input word", or "N is always greater than 100 between the first and fifth iterations". The methods we develop for describing timing relationships will thus have to satisfy these criteria. Furthermore, it will also be true that what is a point in time at one level of description (e.g. a function call), will be expanded into an interval with internal details at a lower level of description (e.g. the model of the function's own behaviour).

1.1.3 LISP Specialized Knowledge

Once an implementation plan has been chosen, it is not necesarilly the case that the code has been determined. Returning to our running example of a hash table, suppose that it was already known that the buckets were being implemented as lists. This would strongly suggest that a plan known as cdring down the list would be suitable for the look up routine. Now this plan is a specific form of a very general plan know as iteration which can have several code realizations in LISP including do-loops, open coded loops using go-to's or even a recursion. Thus the following all accomplish the same task:

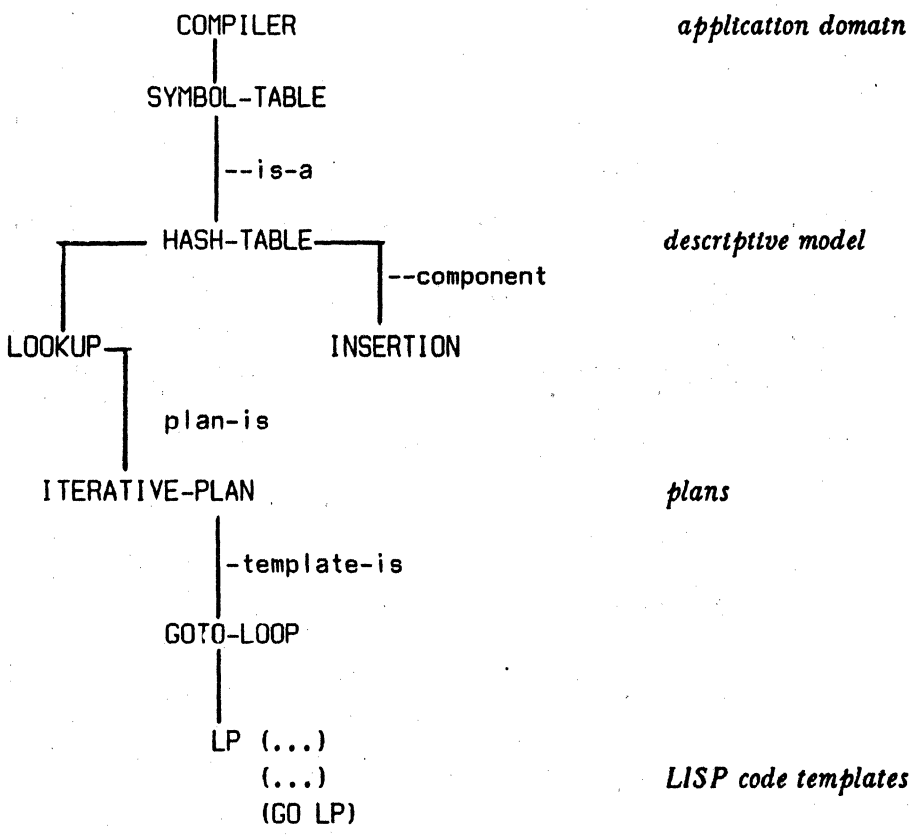
```
(DEFUN LOOKER (LIST ITEM)
  (COND ((EQ (CAR LIST) ITEM) (CAR LIST))
        (T (LOOKER (CDR LIST) ITEM))))

(DEFUN LOOKER (LIST ITEM)
  (DO ((*F (CAR LIST) (CAR *R))
       (*R (CDR LIST) (CDR *R)))
      ((EQ *F ITEM) *F)))

(DEFUN LOOKER (LIST ITEM)
  (PROG (FIRST REST)
    (SETQ FIRST (CAR LIST))
    (SETQ REST (CDR LIST))
```

```
LP (COND ((EQ FIRST ITEM) (RETURN FIRST)))
   (SETQ FIRST (CAR REST))
   (SETQ REST (CDR REST))
   (GO LP)))
```

In addition, each of these pieces of code can be varied in several ways and still exhibit the same behavior. Thus, a third domain of knowledge must be pointed to by the implementation plans, which can roughly be characterized as code level knowledge. Within this domain must be knowledge of the meaning of the various forms of code (particulalry so for fexprs and fsubrs), recognizers for frequenlty used code (cliches), and the ability to infer behavioral similarity at the low level. A great fraction of this knowledge will be represented as templates (or fancy pattern matchers) which can gobble up expected pieces of code, and create models of their behavior and purposes. Thus, although seemingly less profound, this area is absolutely essential to the overall process of program recognition and understanding. It is, after all, fairly trivial to state that in order to understand a large program one must first be able to recognize smaller segments of code as doing something which one already understands.

1.2 ORGANIZATION OF THE KNOWLEDGE BASE

We do not see the organization of the knowledge developing in any sort of strict or global hierarchy. Nevertheless, it is useful to recognize the existence of descriptions at different levels of abstraction, which in reference to a particular locus in the network of concepts can be arranged roughly in layers. For example, consider the following fragment from the description of a compiler:



In this example we see nodes at all levels (and, in fact, we have simplified). The distinctions between layers is most clear between extremes. The lowest level of description is of course the LISP code itself. At the other end of the spectrum, there is the application domain, in which the programmer conceives of his program as the solution to some problem, specified in application

terms (in this case the application is to build a compiler). In going from one layer of description to another there are implicit choices that have been made. For example, the concept of a symbol table in a compiler was here implemented as a hash table, but it could have also been a linear list. Likewise, there are many choices to be made in how to implement the hash table, e.g. bucket hash, overflow tables, linear rehash, etc. At another level, the iterative plan that is used in the implementation of the lookup routine (e.g. in a bucket hash to search down the association list of the bucket), may be implemented either as a goto loop, or as a list recursion. Particular nodes may also occur at differring levels of description, depending on context. For example, the COMPILER node, which is at the topmost level in this local hierarchy, might just be one of several lower level components in a network describing a much larger system.

In order to achieve this desired level of flexibility, it seems necessary to reject strict global hierarchies as a design methodology. Rather, we suggest that the appropriate structure is that of a knowledge network in which it is *a priori* possible to connect any node to any other. However, in order that this does not lead to total anarchy, it is also necessary that any node connect only to a small subset of the nodes of the total network, namely those which it should naturally "know" about. Clearly as new knowledge is added to the network, connections will be added to some nodes, but in general each node will still be directly connected to only a few other nodes. Furthermore, each node may contain several active elements which might well correspond to local strategies for accomplishing certain goals, such as local recognizers, etc. These might call on other such active elements in nodes to which this node is connected (as an example, is-a type connections are naturaly handled this way). Thus, the limitation on the connections at each node, both serves to give the illusion of local hierarchy and to keep control flow within reason

1.2.1 Prototypes and Instantiations

It will often be the case in the knowledge base that a single concept will be used in many different places. For example, our model of an ARRAY could be implicated in the idea of HASH-TABLE if the table is implemented as a LISP array. Elsewhere in a large program which used hashing and also stacks, the ARRAY model could again be used in the context of the implementation of a fixed-size stack. One way to handle this situation is to follow Minsky's Frame Paper <Minsky 1974> and create a prototypical model of ARRAY, to which are attached default instantiations of important features, and any other general knowledge about arrays that the system has, such as typical bugs (e.g. subscript out of bounds), perhaps an example, and eventually natural language processing related information, such as typical lexical realizations.

For each particular occurence of the concept in the context of describing other, perhaps higher level, entities, an *instanttation* of the prototype is inserted. Extrinsic relations between the particular array and the context of use are represented using the instantiation. Typically the prototypical intrinsic description will be shared by all the instantiations, but in the case where more specific or idiosyncratic information is known, this would be attached to the instantiation directly. Thus, when enquiring about the properties of a concept, a search is first made on the local instantiation, and then secondly any unspecified information can be filled in from the prototype, if it exists. There are, of course, much more sophisticated approaches to this general problem of prototypical models and instantiations, but we have no evidence at this time for ways in which ways the simple ideas presented here are lacking for our intended application.

1.2.2 Forking of Models

Because the knowledge base is intended to represent general concepts, there will of necessity be points in the models where choices or *forks* occur. For example, the model of hash tables has such a point in its desciption of implementation plans (e.g. hash-rehash vs. buckets, etc.) It is, therefore, apparent that the P.A. needs to have some coherent philosophy of how to handle alternatives. To this end, we have discerned three classes of forking that occur in the domain of describing programs.

1.2.2.1 Variations

One kind of multiplicity of models occurs when there is one basic form of a concept, either in the sense of being canonical or else some sort of default, but there are also minor variations possible of several features. This is often the case when LISP programmers define their own variations of the standard LISP functions. For example, in one of the programs we were looking at for inspiration, the programmer defined a function MAPCAR2, which was "identical to MAPCAR except that the results of NIL are not included in the final list". In such cases, the most natural form of representation seems to be to consider the relationship between the canonical form and the variations very similarly to the relationship between the prototype and its instantiations, described in a previous section. Thus each variation would refer to the canonical version, and have its own local list of variations and exceptions. Likewise, the canonical form should probably have some indication of the existence of possible variations.

1.2.2.2 Design Choices

There are other cases of forking in which each of the alternatives are of equal stature, such that none can properly be thought of as the root of the others in the sense of the previous section. In all the cases we have come across, the forking occurs on the basis of what can be thought of as one or more design choices. The simplest case is that in which one design choice determines forks at several (one or more) different points in the model. For example, there are four ways one can handle so-called "collisions" in hash tables: (1) rehash, (2) linear search, (3) buckets, (4) overflow table. Given that the choice is made once between these design alternatives (think of it as setting the position of a global switch), which of the alternative behaviours and implementations are chosen at choice points in the insert, lookup, and delete routines is also determined. This might be illustrated in tabular form as follows. Suppose the branches of the forks in the models of the three routines are labelled respectively, F1: a,b,c,d; F2: p,q,r,s; F3: w,x,y,z.

Design Choice	F1	F2	F3
(1)	a	p	w
(2)	b	q	x
(3)	c	r	y
(4)	d	s	z

A slightly more complicated situation would occur if there were several design choices, but as long as they acted independently, i.e. the choice at each fork was determined only by the position of one global "switch", the implementation seems to follow directly from the present exposition.

However, the third and most complex class of forking behaviour results when there is a high degree of interaction between choices at different forks. In this case, even though there may be clear design choices in the mind of the programmer, the decisions are not separable. This kind of

feedback and non-isolatability is a major feature in the domain of electronics design <Sussman and Brown 1974>. However, we have found difficulty finding natural examples of this kind in programming. This may either be a result of the nature of the domain itself, or, more probably, due to the fact that people are not very good at handling the type of reasoning required in such situations, so they avoid them in programming. Nevertheless we can give here an abstract schema similar to the one above to illustrate. Again, consider three forks, F1: a,b,c; F2: p,q,r; F3: x,y,z. The following interactions could hold:

(i) the choice of a determines the choice of either p or q

(ii) the choice of c determines the choice of x

(iii) the choice of z determines the choice of q

Further computation shows that there are only four permissible combinations.

Permissible Comb	F1	F2	F3
(1)	a	p	y
(2)	a	q	z
(3)	b	r	y
(4)	c	r	x

Recast in this tabular format, this class of forking seems to resemble closely the previous case of simple design switches. The point to be kept in mind, however is that the four cases here do not correspond to four simple options of one choice, but result rather, out of the interaction of several choices.

CHAPTER TWO. PROGRAM TELEOLOGY AND ANNOTATION

2.1 The Function of Annotation

As mentioned in the previous chapter, our programmer apprentice's major task will be the construction of a model of the program which it is working on, using concepts from its knowledge base. We follow this approach primarily because we feel that this is what expert programmers do when presented with code with which they are unfamiliar. It has been our observation that this process of program understanding ranges from difficult to the impossible unless various forms of clues, particularly mnemonic names and line by line commentary, are given to the person who is attempting to make sense of the code. Even with these, it is still a non-trivial task to understand the program unless the overall plan and intentions of the code are known. Although commentary on code is in general famously neglected, by studying the comments of various of our colleagues, we have discerned several ways in which people do use annotation to help simplify the process of understanding.

Annotation of code can, in general, be divided into two broad categories. First, there are comments that assume there is available to the reader a knowledge base of information relevant to the code under consideration, i.e. the assumption is that the code is doing something the reader knows about. The other category contains precisely those comments used when this assumption cannot be made. In this case, the comments will attempt to present or fill in the missing background knowledge. Because this second class of commentary is, by definiton, more demanding of the programmer writing the code, it is precisely these comments which programmers most frequently skip. Given that the design of our apprentice requires it to have on hand a large base of background information, it is our hope that programmers might be able to use the apprentice,

without having to pay a huge price in constructing detailed commentary.

In those cases where the programmer is intending a comment to refer to already existy background knowledge, it has been our observation that the most frequent way of indicating this is through the use of mnemonic identifiers. Thus, for example, the lookup routine for the hash table would typically be called LOOKUP, or some variant, rather than say FUNC1. We would expect this practice to carry over into the P.A. environment, and consider this a perfectly valid and quite efficient manner of documenting code. Nonetheless, as any experienced programmer will attest, it does have its problems. Firstly, one may get tired of the burden of having to continually make up "meaningful" names, and since they also tend to be longer than calling variables V1,V2,V3, etc., one also can get tired of the extra typing incurred. Further, often if one later slightly changes the behaviour of a segment of code, the mnemonic names then also have to be changed; or if left alone, they become misleading. Finally, with mnemonic names, as compared to automatically generated unique symbols, there is always the danger of inadvertent duplication.

The second form of commentary which refers to the knowledge base are comments of the form "this is the hash table lookup routine" or "the next five functions make up the hash table". Typically, such comments appear at the "head" of the code, that is, they typically precede a discrete unit of code and appear as an introductory remark. Such comments serve the function of setting context and supplying necessary but unstated information. For example, the comment "this is the bucket lookup" provides among other things a pointer to the *descitptive model* which contains the *specs* for the function on which it appears. Similarly "use the rehash scheme", would have the effect of telling us what implementation plan(s) are being used. Another important function of these comments, as illustrated by the previous example, is that they can choose among the various

possibilities presented by a fork point in the knowledge base. In general, these types of commentary serve the role of aiding the programming apprentice in understanding the program (i.e. in constructing an internal model of its structure and behavior) by selecting the appropriate units of knowledge out of the knowledge base. Because of this we call these *selector* annotation.

The second broad category of annotation involves those circumstances in which the programmer feels that there is relevant information to convey which is not available within the background knowledge. In this case, he is faced with the task of presenting the knowledge on the page. Because the information which he must present is identical to that which would have been in the knowledge base, the comments which the programmer will use to do this will, typically, have a one-to-one correspondance to the types of objects contained in the knowledge base. That is, they will answer how, what, and why type questions by presenting parts of the *specs* and *plans* for the referenced code.

Comments that answer "why" questions, are what we call *purpose* annotation. For example, "positive, so function-23 won't get gronked" or "to make var-10 positive for the square-rooter". These tend to be "side of the code" commentary. What typifies *purpose* comments is that they establish a link between the behaviour of the code upon which they appear and some other segment to which they refer. In this example, the comment informs us that the purpose of the current behaviour of making something positive (assuming we were also reading the code, we would know what the something was), was because the behaviour of function-23 is undesirable if this is not the case. That is to say, this segment is establishing a prerequisite for function-23. Another answer to "why" questions is that the current step is being performed to achieve an overall goal, i.e. it is a main step. We will explain the theoretical framework which this refers to in the

next section.

"How" questions, tend, by and large, to be answered by the selector comments which we mentioned above, but they are occasionally given explicit answers on the page. Again, these tend to be comments too large to be on the side of the code and, therefore, often appear as "top of the code" introductions. When given, they present a teleology for the segment, i.e. a schema of steps and their purposes. For example, the code might have an introduction like "use the hash routine to get a bucket, then use the bucket lookup to get the answer". Again, we will have much more to say about the theoretical framework for this in the next section.

The remaining class of this trilogy, "what" comments, can usually be typified as providing all or part of a a *descriptive model* to the apprentice. We therefore refer to these as *definitional* annotation. For example, in an interactive bibliography program, we observed a half-page comment which explained the structure and use of an entity call a "prompt". The details of this are not relevant here, but it was interesting for us to note that this description included precisely those elements which belong in descriptive models, e.g. *specs*, parts decompostions, etc. It is extremely typical for such definitions to define a data structure which will have limited application, i.e. it is used only in one section of the system and is of relatively little value to future programs. If, however, it defined something of more general value, the apprentice should be able to file it away in its knowledge base, since the information already has the right structure.

As a special case of the above, there is a very common form of commentary, namely stating explicitly the specs of a segment of code. Sometimes these will appear as simple "head of the code" type statements, e.g. "when given a list, returns its third element, if present, otherwise returns

'foobar". More often, however, these specs are broken up into their components, namely pre- and post- conditions. Because segmentation boundaries are often arbitrarily drawn in LISP, these two types of commentary often appear as "side of the code" comments. For example, "assume a negative", or "now the item is in table". The first example is a modal expression, and specifies a condition that is assumed or expected to hold just prior to the execution of the segment of code which it annotates, presumably because the correct behaviour of the code depends on the specified condition. These we will call *expectations*. The second example asserts that a certain condition will hold immediately following (and usually as a result of) the annotated SEGMENT of code. These we will call *assertions*. These two concepts will have an important role in the theoretical framework to be described following.

Finally, just to complete our survey of program commentary, we must pay homage to the incredible diversity and imaginativeness one finds in the annotation of some hackers' programs--everything from sonnets to Pig Latin. We make no claims for our P.A. vis a vis such material. Nonetheless, even in the domain of more idiosyncratic annotation, there are several recurrent forms that bear mention and consideration. The following is a suggestive list: "this is a kludge", "missing code to be inserted here", "this needs to be fixed", etc.

2.2 Theoretical Framework for Teleology

2.2.1 Segmentation of the Code

In order to establish the connection between the raw LISP code and the varied levels of descriptive framework outlined in Chapter 1, we need the notion of *segmenting* the code, that is to say drawing a conceptual box around one portion of the LISP code and speaking of it as a unit with input-

output (or before-after) behaviour. The boundaries one choses to divide up code segments are arbitrary. That is to say, where one thinks the boundary is depends on what one is interested in. Furthermore, any one segment will typically be subdivided internally into smaller segments, with correspondingly more primitive behaviour, as the level of descriptive detail requires. For example, a whole function definition, a group of related functions (obviously not required to appear contiguously in the code listing), or a single form within a function may be thought of as code segments. The only requirement is that they are aggregated for the purpose of describing their net behaviour.

It is also important to realize that the decomposition of a segment will typically not be complete, i.e. some code will be left over when a segment is divided into its logical parts. For example, consider the following code:

```
(DEFUN A
   (PROG ()
      (B ...)
      (C ...)
      (RETURN X)))
```

Here the main segment is A, whose two component parts are invocations of B and C respectively. The PROGN and RETURN statements are what could be called the *connective tissue* between the subsegments of A. As such they carry very important control structure information. The PROGN specifies the temporal sequence between the invocation of the two steps, B and C, and the RETURN determines what the net output of the function A will be.

A second complication in the segmentation can arise when segments overlap. For example, a single line of code could be a natural part of two different contiguous segments, in much the same way that a single resistor in an electronics circuit <Sussman and Brown, 1974> could be naturally

thought of as being simultaneously part of both the output network of one transistor, and the bias network of the next. However, this situation occurs much less often in programming, as compared to electronics, probably because in programming the possibility exists of making independent subroutine calls.

Thus code segments, or *segments* for short, will be the basic formal object upon which the theoretical framework for describing programs is built. It bears emphasis here that in the situation of recognizing and understanding a previously unseen LISP program, the problem of properly dividing the code into functional segments is both difficult and crucial. The analogous problem has been faced in visual recognition research and is as yet unsolved in the general case. We will come back to this problem in Chapter 3 following, where we discuss program recognition and understanding at greater length.

2.2.2 Program Specs

Our notions of how to describe the intrinsic behaviour of code segments follow firstly the rich tradition of input-output specification and more parochially, Carl Hewitt's elaboration of the idea of "contract" in the development of his "actor" formalism. The essential idea is that a given segment of code has certain expectations, incoming assumptions, or *pre-conditions* (we will use the terms interchangeably) that are assumed to hold just prior to execution commencing for that segment, and upon which the correct functioning of that segment of code depends. These are the *input specs* of the segment. Correspondingly, the *output specs*, are a set of assertions, outgoing entailments, or *post-conditions* that are promised to hold just following, and usually as a result of, the correct execution of the code segment. Moreover, the specs are intended to be only the intrinsic description of the code segment, in the sense explained in Chapter One. That is to say, the

conditions in the specs should be a reflection of the internal workings of the code segment, and will be the same regardless of the context in which it appears. The basic syntax of a specs expression is the following:

```
(SPECS  segment-name  ( input-objects ) ( output-objects )
        (EXPECT ( pre-condition ))
        (EXPECT ...
        (ASSERT ( post-condition ))
        (ASSERT ...  ))
```

Thus we see the program specs are essentially a list of clauses of two types, EXPECT clauses, expressing pre-conditions, and ASSERT clauses, expressing post-conditions, preceeded by header information. The header information consists of the segment name, and a list of input objects and of output objects. The input objects are the data structures which are in any sense input to the behaviour of the segment. In particular, any object mentioned in a pre-condition must appear in the list of input objects. In terms of LISP code, the input objects could be formal arguments to a function, if the segment were a separate LISP function, or else just globally available data structures, to which the code segment referred. Correspondingly, any object upon which the behaviour of the segment has a side effect (e.g. changing value, creating a new object) must appear in the list of output objects. The post-conditions will express the (new) properties of the output objects, often referring to some of the input objects to do so. In the LISP code, the output object could be the returned function value, or more generally, a global data structure which was modified. This is all quite general, so let us proceed with an example. The following is the way we currently envisage representing the intrinsic behaviour of say, a segment of code that performs the square root. In order to present the following examples, we have had to choose some details of notation. We have done this in the way we currently find most natural. We certainly expect the notation to change in detail as our research progresses, so that the current version should be taken

"with a grain of salt":

```
(SPECS SQRT (NUMBER-1) (NUMBER-2)
       (EXPECT (GE NUMBER-1 0))
       (ASSERT (EQ (TIMES NUMBER-2 NUMBER-2) NUMBER-1))
```

The first thing to remark about the example is our convention for naming the input and output objects, NUMBER-1 and NUMBER-2, respectively. The choice of local symbol is not arbitrary. Rather, it is intended to carry with it type information about the object referred to. Following the discussion in Chapter One, each object is seen as an instantiation of some prototype, e.g. NUMBER-1 is an object about which we can get more information by referring to the description of the prototypical NUMBER. The notion is that a naming convention is an indirect reference to background knowledge. We will make use of this quite often. For example, in debugging mode, the P.A. might check the implicit (in the notation) expectation that the input was a well-formed NUMBER by applying the LISP predicate NUMBERP to a particular input in question. The information that this was a correct strategy would be part of the knowledge associated with the concept NUMBER. There would also be other information associated with NUMBER, for instance, that it made sense to talk about the SIGN and EXPONENT of a number, and how to calculate them if necessary. The reference to the concept of SIGN is then itself also a potential indirection to more information, e.g. that the possible signs are NEG, POS, and ZERO.

An alternative form of the first of the specs above might take advantage of this implicit knowledge, for example:

```
(EXPECT (SIGN NUMBER-1 POS))
```

Thus, we see that the exact form of the clauses of the specs will be greatly influenced by the deductive mechanisms that will use them. Since our P.A. will use database-like deduction rather than standard theorem proving, it is not surprising that our input-output conditions have the

flavor of PLANNER statements, rather than predicate calculus expressions.

Returning to our ongoing example of hash table programming, let us present what might be the specs of the insert, lookup, and delete routines, respectively.

```
(SPECS INSERT (ITEM-1 TABLE-1) (TABLE-1)
       (EXPECT (NOT (MEMBER ITEM-1 TABLE-1))
       (ASSERT (MEMBER ITEM-1 TABLE-1)))

(SPECS LOOKUP (KEY-1 TABLE-1) (ITEM-1)
       (CASES
        ((EXPECT (MEMBER ITEM-1 TABLE-1))
         (ASSERT (KEY ITEM-1 KEY-1))))
        ((EXPECT (NOT (MEMBER ITEM-1 TABLE-1)))
         (ASSERT (EQUAL ITEM-1 NIL)))))

(SPECS DELETE (KEY-1 TABLE-1) (TABLE-1)
       (ERASE (MEMBER ITEM-1 TABLE-1)))
```

The first thing to notice in the above example is that the specs of the lookup routine splits up into *cases*. This will often occur when we are describing more complicated behaviours. The syntax is intended to mean that each top level clause within the CASE expression is itself a set of input-output specs. For each case, if the expectations are met, then the resulting output conditions may be asserted. The expectations of the code segment as a whole will be considered satisfied if and only if all non-CASE-embedded expectations are met, and the expectations of at least one case in each CASE expression are also met. The examples above also make use of the indirect reference feature in several places. Firstly, in the lookup routine, the input object KEY-1 potentially brings into the context the knowledge associated with the concept of KEY, e.g. that KEY's are part of ITEM's. This relationship would be used by the deductive mechanisms to resolve the referent of ITEM-1 in (MEMBER ITEM-1 TABLE-1) as the ITEM whose KEY is KEY-1. Similarly, in the specs for the delete routine. Finally, the ERASE statement should be noticed in the specs for the

delete routine. This reminds us again that we are using a database deductive scheme, wherein side effects are simulated by manipulating the current assertions in the database.

2.2.3 Purpose Links

Now that we have formalized intrinsic descriptions in our system in the form of program specs, we are in a position to give a formal characterization of the extrinsic relationships between code segments. This is found in the notion of *purpose links*. A purpose link is supposed to reflect the intuitive idea that segments of code are built by the programmer into a "purposeful" (i.e. teleological or goal-directed) structure by the way their input-output behaviours interrelate. Thus we will define a purpose link formally as establishing a *correspondence* (in various senses) between parts of the specs of two segments. The various kinds of correspondences will then give rise to different kinds of purpose links.

On this basis we divide purpose links into two broad classes. Firstly, we have the class of correspondences between the output specs of one segment, and the input assumptions of another. This is called a *prerequisite* link. The simplest example is an identity match between one output clause of segment A and one input clause of segment B, e.g.

```
(SPECS A (...) (ROOT-1 ...)
       (EXPECT  ...)
       (ASSERT (SIGN ROOT-1 POS))
       (ASSERT  ...))

(SPECS B (ROOT-2 ...) (...)
      (EXPECT  ...)
      (EXPECT (SIGN ROOT-2 POS))
      (ASSERT ...))
```

Implicit in the purpose relationship between these two segments of code is thus also the fact the

ROOT-1 corresponds to ROOT-2, i.e. that the programmer intends in his plan for the output object of A to be the input object to B. It is important to remark here that a necessary ordering condition immediately follows from the prerequisite link between A and B, i.e. that the execution of A precede the execution of B. These kinds of necessary conditions will play an important role in the recognizing program structure, as we will describe in Chapter 3.

The second basic class of purpose links is called *main-step* links. This is the class in which there is a correspondence between the output specs of one segment, called the subordinate segment, and the output specs of its superordinate. For example, adding more detail to the specs for hash table insertion: (Note that in following examples, for ease of reading we will omit the unique instantiation identifiers on object names, where there is no ambiguity; i.e. we will simply say BUCKET instead of BUCKET-n, when there is only one bucket in the context.)

```
(SPECS INSERT (KEY DATA) (ENTRY)
       (EXPECT ...)
       (ASSERT (MEMBER TABLE ENTRY))
       (ASSERT ...))

(SPECS BUCKET-INSERT (BUCKET KEY DATA) (ENTRY)
       (EXPECT (EQ BUCKET (HASH KEY)))
       (ASSERT (MEMBER BUCKET ENTRY))
       (ASSERT ...))
```

In this example we envisage the situation where the INSERT segment is itself made up of two subordinate segments. The first is to determine (by application of the hashing algorithm) the appropriate bucket in which to insert the item, and the second, called BUCKET-INSERT, actually puts the item into the chosen bucket. The BUCKET-INSERT code segment thus achieves a main step in the described behaviour of the INSERT segment, of which it is a part. For a main-step purpose link to be meaningful between two segments e.g. B is a mainstep of A, it is a necessary

condition that B is *subsumed* by A. This can take two forms: either the code segment B is explicitly part of the open code of segment A, or segment A contains a call to the function which is segment B.

We have avoided referring to purpose links as *matchings* because in the general case a simple pattern match will be insufficient to link together one segment's outgoing assertions and the incoming assumptions of some other segment. A simple example of this is given in the *SPECS* for INSERT presented above. There clearly exists a main-step link between the BUCKET-INSERT and the INSERT, namely, the BUCKET-INSERT routine achieves the overall goal of INSERT by putting the entry into the table. However, this connection is not a simple syntactic match; BUCKET-INSERT only promises to acheive:

```
(MEMBER BUCKET ENTRY)
```

while INSERT requires:

```
(MEMBER TABLE ENTRY)
```

Clearly, the purpose link is not a simple pattern match, but rather also includes the justifying deduction:

```
((MEMBER BUCKET ENTRY)-->(MEMBER TABLE ENTRY))
```

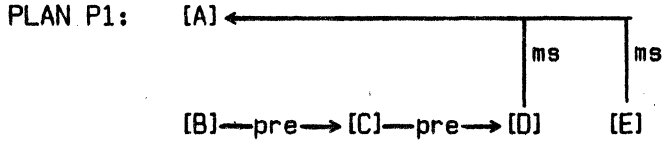
Another type of complication of our simple model is that there are links between segments which do not seem to fit naturally into the framework of *purpose links*. For example, consider a code segment whose job it is to put a red block on a table. Let us assume that the programmer has available a painting routine (which can paint blocks red) and a positioning routine (which will be used to put the block on the table). There is no clear ordering of these steps, i.e the block can be painted and then positioned, or the other way around; let us look at the structure of these routines.

```
(SPECS RED-BLOCK-ON-TABLE (TABLE-1 BLOCK-1) (BLOCK-1)
       (ASSERT (ON TABLE BLOCK-1))
       (ASSERT (RED BLOCK-1)))

(SPECS ON-TABLE (TABLE-2 BLOCK-2) (BLOCK-2)
      (ASSERT (ON TABLE-2 BLOCK-2)))

(SPECS PAINT-RED (BLOCK-3) (BLOCK-3)
       (ASSERT (RED BLOCK-3))))
```

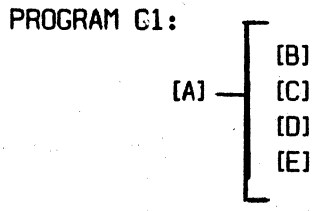
The most obvious links are the two main-step purposes between the output assertions of ON-TABLE and PAINT-RED , and the output assertions of the main routine. Secondly, and more subtly, if the plan is to work it must be the case that the BLOCK which is the output of ON-TABLE is the same as the input BLOCK of PAINT-RED. We will call this type of links a *shared-value* link. These links carry a necessary condition that the code must be arranged in such a way that a value can in fact be shared by the two routines involved. This can be done in either of two ways: either the data structure is available globally to both segments (i.e. it is bound at a higher level), or it is passed as an explicit argument in a function call to the second segment. At the highest conceptual level, therefore, shared value links do not impose ordering but merely syntactic restraints which guarantee that the value may be shared.

2.2.4 Plans

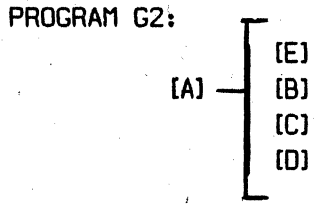
We are now in a position to clarify what we mean by a plan. In the context of the P.A., a plan is defined formally as a *schema* of purpose (and perhaps other) links between segments of code. For example,



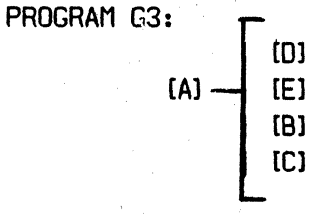
This should be understood as follows: there is a code segment, A, which is achieved as two main steps, D and E. D has a chain of segments, B and C, which are prerequisites. Given this plan of a program, which specifies the important behavioral interactions (e.g. the purpose links) between code segments, it next make sense to consider how the actual surface structure (see Section 3.1 for definition) of the code might be arranged compatible with the plan. For example, program G1 satisfies the necessary conditions implied by the purpose links:



So would



However, G3 could not be a possible implementation of plan P1, because the prerequisites of segment D do not preceed it in the actual code.



A plan may be specified to varying degrees of detail, in two different senses. Firstly, plans may be nested within plans because what is called a segment (i.e. "box") at one level of description (or planning), can itself have internal structure, made up of sub-segments interrelated by their own plan. For example, the top level segment A in plan P1, could (and will typically) itself enter in as a

component of some larger plan.

Along another dimension, a plan can be further particularized by specifying the form of the actual input-output spec clauses that enter into the purpose links. Thus for example, rather than just saying there is a prerequisite link between B and C in plan P1, we might be more specific, for example:

```
(PREREQ (B. (ASSERT (GE ROOT 0)))
        (C. (EXPECT (NOT (SIGN ROOT NEG)))))
```

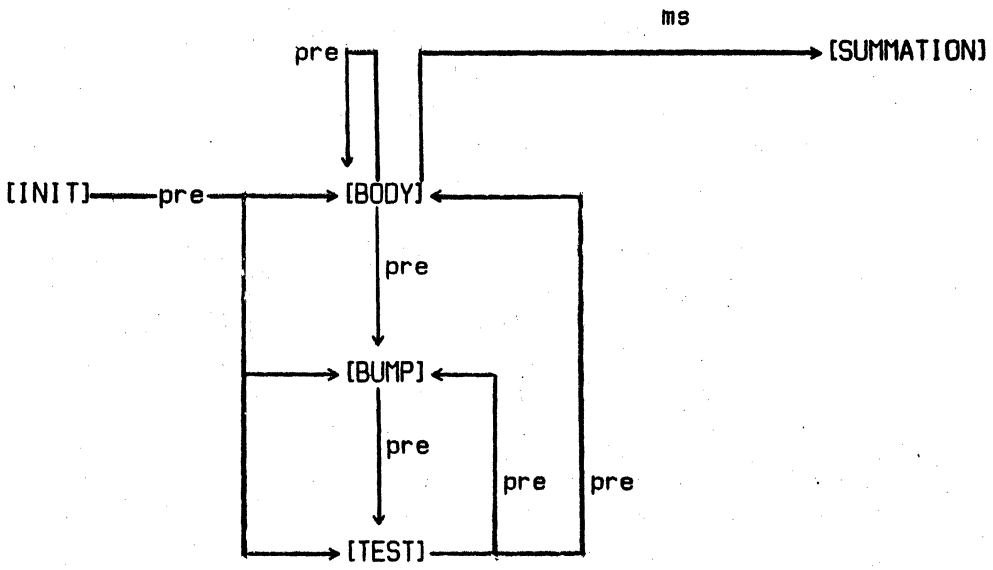
This says specifically that there is a prerequisite link between the output assertion of B that (GE ROOT 0) and the input expectation of C that (NOT (SIGN ROOT NEG)). Note again that, in fact, such a link references the deductive fact that (ge x 0) implies (not (sign x neg)) which would be in the descriptive model of SIGN. Depending on the state of knowledge of the P.A., the plans that are its current attention will vary in both these senses of detail. At one end of the spectrum, the plan for a current user's program will be highly detailed and particularized, so that enough information is immediately available for programming assistance. Leaving aside for the moment the question of how the plan becomes particularized, let us give an example of how we think the complete plan might look for a simple program. Suppose the user had a program to sum the numbers from 1 to 10 using an iteration. The specs of the whole program viewed as a segment are thus:

```
(SPECS SUMMATION () (SUM)
   (ASSERT (EQ SUM (SIGMA (I 1 10) (I)))))
```

where the SIGMA expression has the obvious syntax. The internal structure of the program then consists in this case of the four standard blocks in an iterative plan: INIT, BODY, BUMP, and TEST. They have particular specs as follows.

```
(SPECS INIT () (SUM CTR)
        (ASSERT (EQ CTR 1.))
        (ASSERT (EQ SUM 0.)))

(SPECS BODY (SUM-1  CTR) (SUM-1)
        (EXPECT (EQ (SUM-1 (SIGMA (I 1 (SUB1 CTR)(I)))))
        (EXPECT (LE CTR 10.))
        (ASSERT (EQ SUM-1 (SIGMA (I 1 CTR)(I)))))

 (SPECS BUMP (CTR-1) (CTR-1)
         (EXPECT (LE CTR-1 10.))
         (ASSERT (EQ CTR-1 (PLUS CTR-1 1.))))

 (SPECS EXIT-TEST (CTR) ()
       (CASES
         ((EXPECT (LE CTR 10.))
         ((EXPECT (GT CTR 10.))))
```

First let us give the basic plan schema which this program follows. Then we will discuss each purpose link in detail. The plan is:



Perhaps the most important purpose link in this plan is:

```
(MSTEP (BODY.(ASSERT (EQ SUM-1 (SIGMA (I 1 CTR)(I))))
       (SUMMATION.(ASSERT (EQ SUM (SIGMA (I 1 10) (I)))))))
```

This, together with a deductive fact about SIGMA (namely that sigma from i to j of f(i) is equal

to f(j) plus sigma from i to j-1) says that each iteration of the body in some sense achieves a main step of the superordinate segment. The required deductive fact which allows the match would be associated with the descriptive model of SIGMA in the network of background knowledge. Furthermore, the match implies that the final output object of SUMMATION, ie. SUM, is the same SUM as appears in the output of each step of the BODY.

```
(PREREQ (INIT.(ASSERT (EQ CTR 1.)))
        (BODY.(EXPECT (LE CTR 10.))))

 (PREREQ (INIT.(ASSERT (EQ CTR 1.)))
        (BUMP.(EXPECT (LE CTR 10.))))

 (PREREQ (INIT.(ASSERT (EQ CTR 1.)))
        (EXIT-TEST.(CASE (EXPECT (LE CTR 10.)))))
```

The above prerequisite links express the fact that the purpose of the INIT segment is to satisfy the input expectations of the other three segments (at least on the first iteration). Notice that the match here is between the assertion (EQ CTR 1.) and (LE CTR 10.). This points out one of the kinds of 'smarts' the deductive and pattern matching mechanisms must have. Furthermore, notice that in the last link it is specifically indicated that it is a CASE of EXIT-TEST that is involved. It will turn out that the case structure of program specs will carry a large part of the descriptive power of the formalism we have developed for plans. For example, the other case of the specs for EXIT-TEST is satisfied by the following prerequisite link:

```
(PREREQ (BUMP.(ASSERT (EQ CTR (PLUS CTR 1.))))
        (EXIT-TEST.(CASE (EXPECT (GT CTR 10.)))))
```

Here we see some rather sophisticated reasoning implicit in the matching of conditions. Firstly, within the specs for the BUMP itself the interpreter (a noncommital word) needs to distinguish between the old and new values of the CTR. The clues to this are in the fact that CTR is both

the input and output object of BUMP. Secondly, given the expectation of BUMP that (LE CTR 10.), a possible value to substitute for CTR in (PLUS CTR 1.) is 10., giving (BUMP.(ASSERT (EQ CTR 11.))), which then matches with the case (EXPECT (GT CTR 10.)) in the EXIT-TEST. To summarize, this prerequisite is the condition that the iteration eventually terminate.

```
(PREREQ (EXIT-TEST. (CASE (ASSERT (LE CTR 10.))))
        (BODY. (EXPECT (LE CTR 10.))))

(PREREQ (EXIT-TEST. (CASE (ASSERT (LE CTR 10.))))
        (BUMP. (EXPECT (LE CTR 10.))))

(PREREQ (BODY. (ASSERT (LE CTR 10.)
        (BUMP. (EXPECT (LE CTR 10.))))
```

These prerequisites establish the basic iterative framework; that is to say, given the failure case of the EXIT-TEST, the prerequisites are then satisfied for another iteration through the BODY and BUMP. One should notice here that the clause (ASSERT (LE CTR 10.)) does not explicitly appear in the specs of EXIT-TEST or BODY. To be pedantic, the specs for these segments should have been written:

```
(SPECS EXIT-TEST (CTR) (...)
        (CASES
          ((EXPECT (LE CTR 10.))
           (ASSERT (LE CTR 10.))  ...etc.

(SPECS BODY (CTR ...) (...)
        (EXPECT (LE CTR 10.))
         ...
        (ASSERT (LE CTR 10.)))
```

However, it seems quite reasonable to assign to the interpretative and deductive mechanisms of the P.A. the responsibility to automatically generate such redundant assertions from a general rule. The rule would state that any input condition is automatically an output condition if none of its terms appear in the list of output objects (i.e. there are no side effects on any of the terms). In

fact, exactly this would happen in any PLANNER-like language, since assertions will not disappear unless explicitly erased.

Now that we have given an example of a particular plan in gory detail, let us move to a higher level and acknowledge the existence of broad *classes of plans*. For example, the particular plan presented here is only one member of a class of many possible iterative plans. All the members of this class have the concepts of INIT, BODY, BUMP, and TEST in common, but may use them slightly differently in particular instantiations. For instance, the BODY, BUMP, and TEST might be implemented in various orders. More profoundly, it is true that all iterations implicitly refer to some total ordering of the items being iterated over (numerical, list position, etc.); which particular total ordering is use is peculiar to each individual iterative plan. Other examples of classes of plans are recursive plans, linear plans, and dispatch plans. We are not certain at this point in the research how best to capture the shared properties of these classes of plans. Perhaps it will be possible to represent in the same formalism a prototypical member of each class. More likely, however, there will probably be some cluster of expertise in the knowledge base having to do with each class of plans, which will be applied to members of the class as appropriate.

Finally, let us summarize here by saying that plans are very much the central notion in the whole operation of the P.A. The plan will form the core of the model that is built of a particular user's LISP program. Thus we expect a significant portion of our research effort to go into learning how to represent plans in such a way that they first, naturally follow the way programmers organize their code and then second, can be utilized by the P.A. in order to perform its various services.

2.3 Annotation and the P.A.

Since the P.A. will be one of a new breed of systems programs that are prepared to deal with program commentary and annotation for purposes other than pretty-printing on code listings, the first step is to develop mechanisms whereby the traditional barriers between code and commentary are broken down. As far as the P.A. is concerned, both will be clues to understanding a program. Since annotation will play such an important role in the P.A. scenario, we might also seek to implement code annotation in a way to improve where possible on the current method. A major difficulty with traditional comments is that they are forced to appear linearly throughout the code, which means they refer implicity by their position to the immediately following segment of code, and by explicit mention to arbitary other segments. This suggested to us a general scheme for storing and accessing program comments, which we implemented as follows. Firstly, in order to facilitate "walking around" in the code, we back-pointered all the code list structure in the program, so that from any point it is possible to find out what higher level expression it is embedded in. Secondly, we implemented the ability to refer explicitly (by pointer or by giving it a unique name) to any list structure or substructure in the code. Using this, all the commentary can be stored and indexed in such a way that a single annotation may refer to many segments of code (e.g. "these are the error recovery routines"), and conversely it is possible to determine, for any segment of code, all the items of annotation which refer to it. We believe this should be a good and flexible framework in which the P.A. will be able to do its work.

We have assumed to a large extent that the programming apprentice will be using a large pre-established knowledge base to which it incrementally adds new knowledge. Because of this, we have concluded that most of a user's commentary could be put into the form of *selectors* and *mnemonic identifiers*. In addition, we have assumed that the user will include the more complex

forms of commentary to the same extent as he does now. However, to be honest, it must be admitted that the average programmer controls a body of knowledge of sufficient size that it would require us several man years of effort and several additional computers of storage to be able to present this knowledge to the programmer as a unified whole. Furthermore, we would initially require the programmer to formalize his commentary to allow the apprentice to understand him. It therefore becomes implicit in our discussions of annotation here, that we expect (or will require) programmers to change their behaviour somewhat when interacting with the P.A., in terms of the character (and perhaps quantity) of their comments. One might then ask how difficult will it be to get programmers to conform to the restricitons of the apprentice environment. This would seem to depend on how good a helper the apprentice turns out to be. We believe that if a P.A. is really successful in helping the programmer with his work, there will be no problem getting him to provide enough annotation to make it possible

CHAPTER THREE. RECOGNITION AND UNDERSTANDING

3.0 Introduction

In the previous two chapters we have been engaged in the business of building up the knowledge base and descriptive formalisms necessary for the P.A. to represent programs at various levels of abstraction. The central topic of this chapter will be to describe how the link-up is made between the P.A.'s rich knowledge base, on the one hand, and a user's particular LISP program, on the other. As a prelude to this important discussion, we wish in this section to move the focus of description back to the code level. Here we wish to ask what kinds of descriptive concepts will be required at the code level, separating this, for the moment, from the mechanisms and processes by which the actual final description is generated. As an aid to this, let us suppose the following highly simplified (and unworkable) two-stage model of the recognition process:

> (STEP 1) The so-called *surface structure* of the program is generated from the code bottom-up by a super indexing program, which utilizes only knowledge of LISP syntax and the semantics of the basic LISP functions, such as PROG, COND, EVAL, etc. What information this surface structure analysis of the program might yield is the topic of Section 3.1.
>
> (STEP 2) The surface structure is merged into a larger model of the program built up by the P.A. from its store of descriptive models and plans on the basis of the commentary supplied by the programmer. The key feature of the merging is that the correspondence is made between the formal segments of the abstract description of the program and actual segments at the code level. A more realistic treatment of building this complete model of the program is the topic of Section 3.2.

Before going on, let us reiterate that this description of the recognition process should be taken only as an item of pedagogy, used to introduce the issues. The two-stage scheme will not work in practice, for several reasons. Firstly, it entirely finesses the segmentation problem, discussed in Chapter Two. Unless the segmentation into functions, PROG's, etc., at the code level fortuitously matches exactly the groupings coming down from the higher levels of description, the merging

process of STEP 2 becomes extremely messy. Secondly, the two-stage process above artifically, and critically, separates the program description into two aspects (syntactic in STEP 1 and semantic in STEP 2, approximately), ignoring the important fact that clues to both kinds of information come from several shared sources: the raw code itself (including mnemonic identifiers), the accompanying annotation, and all the P.A.'s background knowledge about programming as implicated from the other two sources. Any realistic solution to the recognition problem will have to take advantage of this heterarchy.

3.1 Surface Structure in Programs

The major job of the surface structure representation of code is to show the control relationships between segments of code. This information is potentially derivable simply from the nesting syntax of LISP, the rules of the LISP interpreter, and the semantics of the special LISP functions like PROG, COND, AND, OR, etc. At this surface level in LISP, there are only two basic execution time relationships that can hold between code segments. (Of course, at higher levels of abstraction from the code, more complex kinds of control relationships can and will be expressed). The first basic surface structure relationship is the *invokes*, or calling relationship, A --inv--> B, which means that the behaviour of B is invoked as a subpart of the behaviour of A. In terms of execution sequence, this is:

(enter A),(enter B),(exit B),(exit A)

This relationship can result either from an explicit function call in A to a segment B, or the body of B might appear as open code in the body of A. The second basic relationship is the *next*, or sequential relationship, A --nxt--> B, which simply means that the execution of segment B does (or can, depending on branching tests) immediately follow the execution of segment A, i.e.

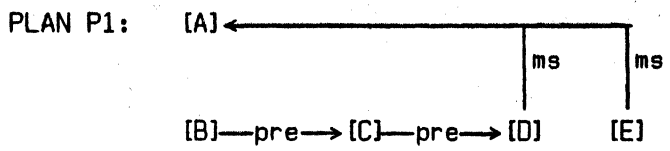
(enter A),(exit A),(enter B),(exit B)

The generalized notions of sequentiality, A --nxt*--> B, and indirect invocation, A --inv*--> B, derive of course from the transitivity of "next" and "invokes", in the obvious way. Finally, for completeness, we should mention the existence of a third candidate execution relationship, i.e.

(enter A),(enter B),(exit A),(exit B)

This is the case of unconstrained coroutines, which cannot be implemented in the basic semantics of LISP.

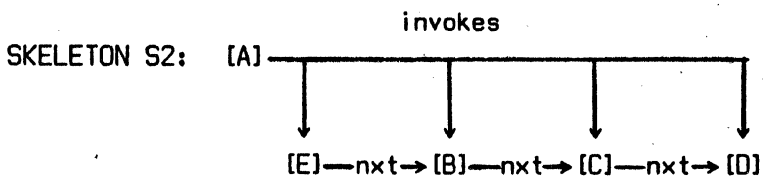
To illustrate our notion of the surface structure representation of a program, let us consider again the plan example from Chapter Two. The following is called the *skeleton plan* of the program because only the type of purpose links between segments is indicated (e.g. mainstep, prereq). A more complete plan would show in addition the specs of each segment, and which assertions and expectations entered into the various relationships.



Now let us take an actual LISP program which is an implementation of plan P1. For simplicity, assume each of the segments, B, C, D, and E have been implemented already as separate LISP functions. The code for the program is then:

```
(DEFUN A (...)
         (PROGN
            (E ...)
            (B ...)
            (C ...)
            (D ...)  ))
```

Using the notation developed above, the *skeleton surface structure* representation of this program would be:



This representation of the program is immediately useful for comparing with its skeleton plan. In particular the P.A. can verify that the necessary ordering conditions implied by the prerequisite and mainstep relationships in the plan are satisfied in the surface structure, i.e.

PLAN	SURFACE STRUCTURE		
[D]--ms->[A]	[A]--inv->[D]		
[E]--ms->[A]	[A]--inv->[E]		
[B]--pre->[C]	[B]--nxt*->[C]	via	[B]--nxt->[C]
[C]--pre->[D]	[C]--nxt*->[D]	via	[C]--nxt->[D]

Note that the surface structure relationship [E]--nxt->[B] is superfluous as far as the plan is concerned. This is quite typical in program analysis, and is simply a reflection of the fact that some details of the code arrangement are not constrained by the underlying plan.

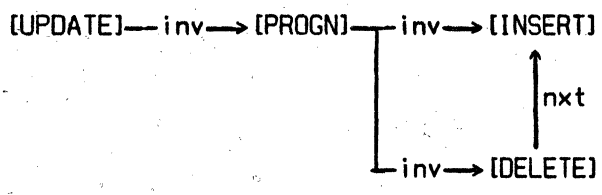
It is very important to realize why the P.A. must have models of the semantics of the special LISP functions like PROG, COND, etc. in order to derive the surface structure of programs. The nesting syntax of LISP is not enough. To bring this out, consider the following two programs, which are syntactically parallel, and yet have very different surface structures:

```
(DEFUN UPDATE (DATA KEY)
   (PROGN
      (DELETE KEY)
      (INSERT DATA KEY)))
```

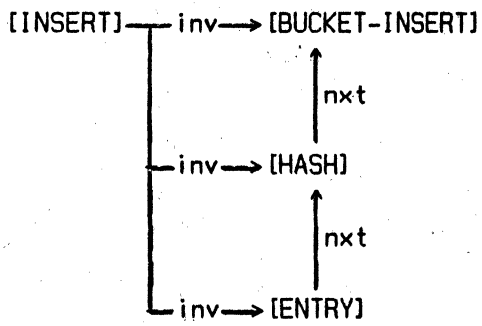
```
(DEFUN INSERT (DATA KEY)
   (BUCKET-INSERT
      (ENTRY DATA KEY)
      (HASH KEY)))
```

In the case of the UPDATE program on the left, the P.A. must have in its knowledge base the information that PROGN is a special kind of LISP function (a FEXPR), which does not follow the usual rules of evaluation. Rather, the plan for PROGN is to evaluate each of its argument

forms in sequence. The P.A. can then derive the correct surface structure skeleton for UPDATE, i.e.



It is interesting to contrast this with the surface skeleton of the INSERT program on the right. On the right, the function syntactically parallelling PROGN is BUCKET-INSERT, which is a normal user-written function (i.e. an EXPR). Thus the normal rules of LISP evaluation apply in deriving the surface structure. First the arguments to a function are evaluated in left-to-right order, and then the function is invoked. According to these semantics, the surface structure comes out quite differently:

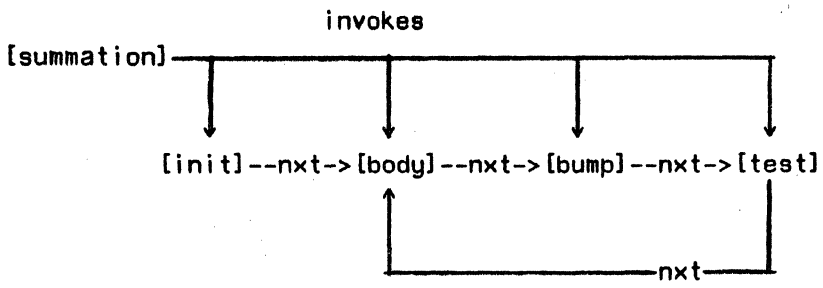


The surface structure analysis is yet incomplete. In addition to the control flow between segments, a super LISP indexer would be able to extract from code some information about data structure (e.g. variable) use. For example, current LISP indexers keep track of what level atoms are bound at, when they are read-referenced, and when their values are changed. The knowledge required to extract this information from code includes knowing the semantics of LISP lambda-binding in general, and specifically the input-output specs of basic LISP functions that lambda bind, such as

PROG, DO, etc., and of the basic LISP functions that can modify data structures, such as SETQ, RPLACA, RPLACD, etc. As mentioned in Chapter One, this information would be part of the P.A.'s knowledge base. To develop this further, let us reconsider the summation example from Chapter Two, this time giving an actual LISP implementation of the plan.

```
[summation]
-----------------------------------------------------
|    (DEFUN SUMMATION ()                            |
|       (PROG (SUM CTR)                             |
|         ------------------                        |
|         | (SETQ SUM 0)   |  [init]                |
|         | (SETQ CTR 1)   |                        |
|       --------------------------------            |
|       | LP (SETQ SUM (PLUS SUM CTR))  |  [body]    |
|       --------------------------------            |
|         | (SETQ CTR (PLUS CTR 1)) |  [bump]       |
|         --------------------------------          |
|         | (COND ((LE CTR 10) (GO LP)))  |  [test]   |
|         --------------------------------          |
|           (RETURN SUM) ))                         |
-----------------------------------------------------
```

The skeleton of this program is:



This skeleton is quite similar to the first example of this section and it could similarly be verified against its plan, which is given in Chapter Two. We will not do that here. Rather let us use this example to develop some new aspects of surface structure. The "invokes" arrows in this example are a reflection of the embedding in [summation] of the open code for its subsegments. The four

basic "next" arrows between the subsegments, [init], [body], [bump], and [test] are simply a reflection of the explicit order of their appearance in the code for a sequential machine. The relationship [test]--nxt->[body], is more interesting, however. It is a result of understanding the meaning of the code (GO LP) at the end of the [test] segment, and the tag conventions for PROG.

In addition to basic control structure relationships, the surface structure representation of a program like this might also contain whatever information about input and output objects of a segment can be derived from a simple analysis of the syntax and basic LISP semantics of a program (i.e. not using any background knowledge about the programmer's higher level intentions). This could be done most naturally by adding to each segment in the control structure skeleton, a *skeleton specs*, which indicates at least the input and output objects of the segment, as derived from the indexer's analysis, and possibly some simple expectations and assertions transferred up to a segment from the specs of the basic LISP functions within it. In the present example, the P.A. might proceed as follows (if it couldn't bring in all this knowledge ready-made from its descriptive model of summations and loops). From the code construct (RETURN SUM), which is part of the important connective tissue between the subsegments of the SUMMATION program, it might conclude

```
(SPECS SUMMATION () (SUM))
```

Looking at the [init] segment, the P.A. might conclude directly from the SETQ's that SUM and CTR were output objects, and that there were no input objects. Thus,

```
(SPECS INIT () (SUM CTR))
```

Reasoning similarly for the other segments, the skeleton specs to be included in the surface structure representation would be:

```
(SPECS BODY (SUM CTR) (SUM))
```

```
(SPECS BUMP (CTR) (CTR))

(SPECS TEST (CTR) ())
```

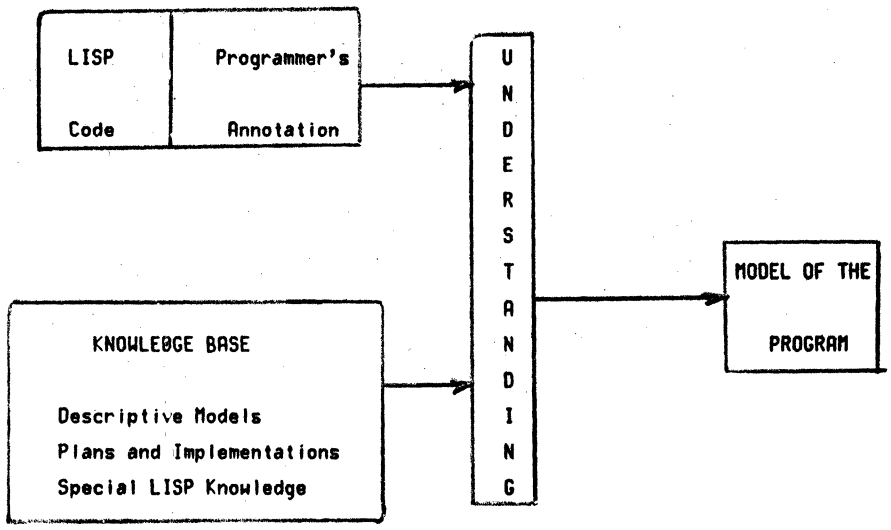
This completes our description of surface structure information about a program. Let us repeat here that the forgoing discussion should *not* be taken to imply that a complete surface structure representation is to be generated by the P.A. by the methods indicated, as a first step towards understanding the program. Rather, this is the kind of information the P.A. can extract from the raw code as a last resort analysis in the absence of strong guidance from its background knowledge base .

3.2 Building the Model of the Program

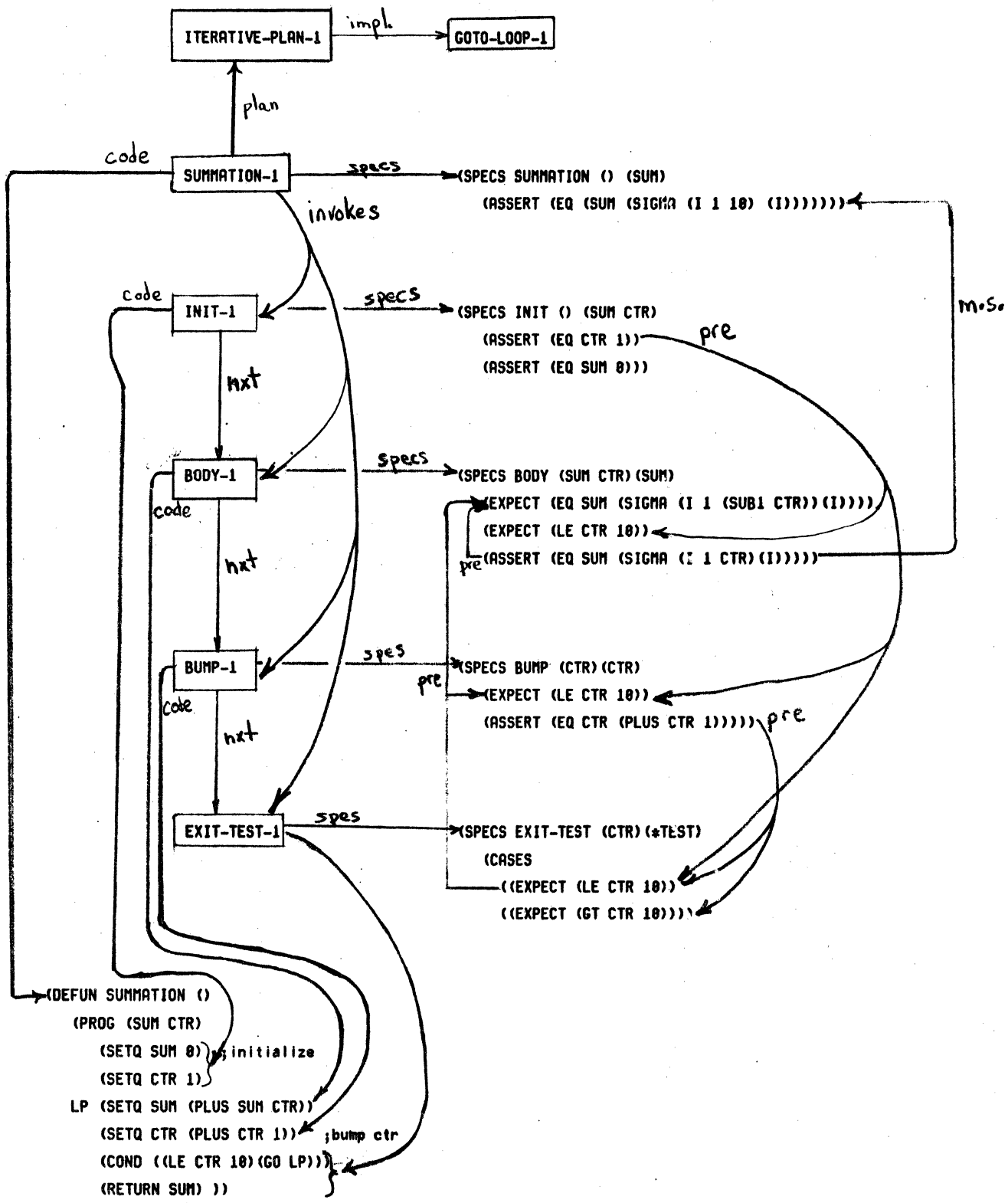
In this section we will undertake to give a more realistic account of how the P.A. would go about understanding a LISP program it had never seen before. We will do this in two steps. First we will define the input-output conditions of the recognition and understanding task we wish the P.A. to perform. Then we will give an informal scenario that reflects our current notions of what the intervening processing might look like. Our scenario will of necessity be quite loose, since discovering exactly how to do the understanding is the major problem we are proposing to research.

3.2.1 Definition of Understanding

At the level of a simple block diagram, the recognition and understanding process could thought of as follows:



Thus, in the context of the P.A., *we will define understanding as the process and result of building a complete model of the program.* The *model* of a program is a complex data structure that describes the program in many ways. Please refer now to the figure on the following page, which gives an example of a complete model.



Model of Summation Program

This figure gives the model for the summation program, which we have been using as a illustration in several previous sections. We have chosen to use it as the example in this section on understanding because it is much smaller and simpler than the full hash table system we have been using as our primary illustration elsewhere. This allows us to give the reader a better feeling for what the complete analysis of a program might look like. The model of the program, as you can see, pulls together all the various kinds of representation we have been developing for program description:

(1) The model includes the *teleology* of the program, represented in terms of the aggregation of the code into segments with associated specs, and their interlation in terms of plans and purpose links.

(2) The model includes the *surface structure* of the program, as represented by "next" and "invokes" arrows between segments.

(3) The model relates segments of the program to descriptive models in the *knowledge base* via prototype-instantiation conventions, e.g. a particular summation program is recognized as an instance of summation programs in general, so that any knowledge the P.A. has about the prototypical entity is available to help deal with the present case. The same holds true for lower level concepts also; for example if a variable is called a "counter" in the model, this implicates all the knowledge the P.A. has compiled about counters in general.

(4) The model of the program reflects the *destgn chotces* that have been made in the particular program. For example, in the P.A.'s knowledge base there are listed several possible implementations for iterative plans (DO-loops, recursion, GOTO-loops). In the model describing this program however, only the alternative that was actually chosen in this instance is shown (i.e. GOTO-loop).

(5) Finally of course, the model also includes the actual *code and annotatton* that comprises the program being described.

On the input side of the understanding process, we have:

(1) The raw LISP code.

Clues to what the programmer is trying to do are buried here in the form of the code itself, and also often in the choice of mnemonic identifiers used to name functions, variables, etc.

(2) The programmer's annotation.

Here the programmer hopefully makes his intentions more explicit. In the ideal case, this annotation would simply be comments in natural language, made as the programmer saw appropriate as reminders necessary to himself or a colleague who will read the code at some later time. More realistically, for the first generations of apprentice, the annotation will have to be in some convenient and hopefully natural-feeling formal language. As for the content, it is a research question to what degree it will have to be more pedantic and extensive than would be appropriate for say, a human assistant. Also, this issue is closely related to how well the P.A. uses its knowledge base to facilitate understanding.

(3) The knowledge base.

(a) Descriptive models.
These specify what is conceptually related to what else, so that the P.A. may generate expectations about what to look for in the code. For example, from the descriptive model of hash tables, the P.A. has the expectation to find an insert routine, lookup routine, etc. Similarly, once the P.A. has discerned that an iterative plan is being used in a program, it then knows to look for an initialization, body, bump, and test. This kind of guidance in what to look for is crucial to the P.A.'s success in understanding the program. (And indeed, the same is true for people, to a great extent.)

(b) Plans and implementations.
Encoded here is the P.A.'s knowledge about standard forms of program control structure, and how they may be implemented in LISP. Here again, the P.A. gains power by often knowing ahead of time what the design alternatives were, so that recognition becomes only a case of determining which choice was made in a particular program.

(c) Special LISP knowledge.
Of course, in order to analyze the code in detail the P.A. has to know the basic syntax of LISP, how the interpreter works, and the semantics of the basic LISP functions. Also in this area of the knowledge base are templates which help span the gap between the lowest level implementation plans, and the particular programmer's idiosyncratic LISP code segment.

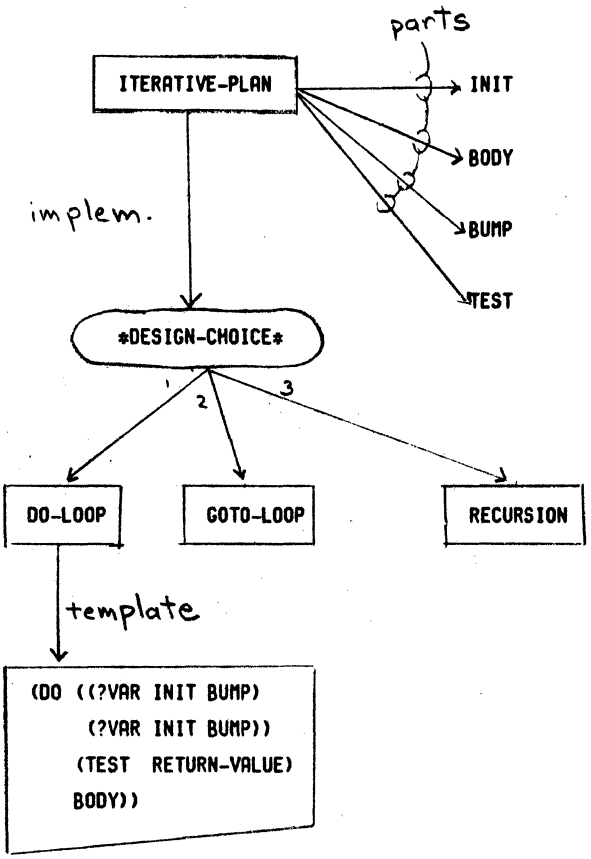
3.2.1 Scenario for Understanding a Program

We are now in a position to present a scenario of what we expect the internal behaviour of the P.A. to be when it is trying to build up a model of a program it has never seen before. Important aspects of this behaviour to pay attention to are the order in which parts of the program are

recognized, and how the various sources of information interact. However, it is also important to keep in mind that this scenario for understanding the summation program is intended only to give a "feel" for the kind of processing and reasoning that we believe will take place. Firstly, it is certainly the case that the exact details of order and method by which the parts of the program are recognized in this scenario are not canonical--we intend them only to be plausible and suggestive of how we think things should happen. Secondly, this one example, in its intentional simplicity and small size, will inevitably miss many important issues in the recognition problem. Third and finally, the plain fact is that we don't know yet exactly how the recognition processes will operate--as stated before, that is part of our research problem. In any case, let us get on with it. Here again is the the program, with its associated meagre annotation:

```
(DEFUN SUMMATION ()
   (PROG (SUM CTR)
      (SETQ SUM 0)  ;initialize
      (SETQ CTR 1)
   LP (SETQ SUM (PLUS SUM CTR))
      (SETQ CTR (PLUS CTR 1))   ;bump ctr
      (COND ((LE CTR 10)(GO LP)))
      (RETURN SUM) ))
```

The first major break in understanding this program is to put it in the context of the appropriate descriptive model. In this case, the programmer has provided this information in the way he named the function, i.e. SUMMATION. Assuming the P.A.'s lexical knowledge of English was adequate, this would immediately invoke the descriptive model of summations. If the function was not mnemonically named (at least as far as the P.A. could understand), the P.A. could alternatively ask the programmer explictly, "what is this program all about?". Now, the descriptive model of summations would include something like the following fragment:



Given that the program is a summation routine, the P.A. thus knows it must have an iterative plan, composed of an init, body, bump, and test. In order to find the code segments that play these roles, however, it first has to determine which of the possible implementations was chosen: do-loop, goto-loop, or recursion. One way for this to happen would have been if the programmer had a comment at the beginning of the code something like ";using a goto loop". Alternatively, each of the design alternatives knows enough to be able to look at the code and see if they were chosen. For instance, the recursion choice might simply look to see if there is a recursion relationship, A --inv*--> A, in the surface structure of the code. Similarly, the do-loop recognizer needs simply look for the surface syntax of the LISP DO construct. Finally, in this example, the goto-loop recognizer would succeed by noticing the PROG construct and the (GO LP) statement. The next major

hurdle is now to figure out how to divide up the code into segments. Sometimes this can be quite easy. For example, consider the following alternative summation program implemented using the DO construct.

```
(DEFUN SUMMATION ()
   (DO ((CTR 1 (PLUS CTR 1))
        (SUM 0))
       ((GT CTR 10) SUM)
       (SETQ SUM (PLUS SUM CTR))))
```

Here we have an example of a "quick kill". From the template associated with the DO-loop implementation in the knowledge base shown above, we can immediately break up the code into segments by simple pattern matching to the syntax of the DO construct. In our primary example, using the GOTO implementation however, it is not so easy. One way to begin is to use the fact that the initialization segment of an iterative plan must always appear first in order at the surface level. Thus, by the way, the ";initialize" comment on the first line of the PROG in the example program is superfluous to the P.A., given that it knows the general fact about the position of initializations. Now that we know where the init segment starts, we need to figure out where it ends. The clue to this is to recognize what part of the code lies inside the iteration loop. The P.A. has already recognized the (GO LP) statement at the bottom of the iteration, from which it is a short step to recognize the the tag LP above delimits the beginning of the iterated code. Since the initialization is not supposed to lie inside the iteration, we have now established that the end of the initialization segment is delimited by the LP tag. Now, within the iteration loop, we have to find the body, bump, and test, which can be varied in their order. However, each have their identifying features. For a start, the exit test segment expects to contain some LISP control primitive, such as COND, AND, or OR, with a GO embedded in it. This can be immediately recognized as the COND clause second line from the bottom. That leaves the bump and body to be accounted for. In this example, we have the helpful comment ";bump" on the fourth SETQ in

the code. The P.A. would then assume (correctly) that the bump segment started there and extended until the beginning of the exit test COND. Alternatively, if the comment had been absent, it seems reasonable to expect the P.A. to have figured out for itself what was the bump segment, by noticing that it was the only line of code which reset the variable CTR (assuming it had already realized the semantic significance of the variable name). As a last resort, of course, the P.A. could always enquire of the programmer where the bump step occurred. Having recognized the bump step, we are essentially finished with the segmentation problem, since all that is left over is the body of the loop, which must be the remaining code inside the iteration.

Now that the code is properly divided into segments the P.A. will complete the model by filling in the specs for each segment, and the relationships between segments, both teleological (purpose links) and surface structure. The surface structure relationships are the easier of the two. In this case the "next" and "invokes" arrows would be immediately filled by some standard algorithm operating on the surface syntax and semantics of the code, as implied in Section 3.1. Following that, the problem of filling in the full details of the specs and the purpose links could be attacked two ways. One way, which would probably work quite well in this simple example, would be to first calculate all the input-output conditions of each segment and then the purpose links between them. directly from some standard algorithm applied to the raw code. However this would not be typical of how we think the P.A. should operate. More realistically, in the descriptive model for each segment type in the plan would be a skeleton or template set of specs that only needed to be adjusted slightly to fit an instantiation of such a segment type appearing in actual code. For example, the prototypical bump segment has an input expectation about the old value of the counter variable, and an output assertion about the new value. It only remains for the P.A. to determine in a particular program, which variable is the counter, and what the value ranges are.

This applies similarly for the other segments. Morever, these prototypical segments in the knowledge base are themselves related together by purpose links into plans. Thus rather than doing pattern matching and deduction on the user program's specs in order to figure out the plan, most or all of the plan comes from the knowledge base along with recognizing the segment types. For example, it is represented in the knowledge base that there is a prerequisite link between the bump and the test, the text and the body, a mainstep link between the body and the invoking segment, etc. Again, as in the case of the specs, only the details have to be adjusted to the program at hand. This completes our scenario of how the model is built. We now claim that the P.A. *understands* the program.

3.3 Control Structure and Implementation Issues

Several words and phrases, for example, "invokes", "recognizers", and "identifying features", which seemed natural to use in the preceeding scenario, suggest certain kinds of control structures that would be appropriate for the P.A. In trying to evaluate these various recognition paradigms of currency in A.I. it is useful to lay them along a dimension, which at one end might be called "hypothesize and jump", and at the other extreme, "wait and see". These contrasting approaches might be exemplified respectively by Minsky's Frame <Minsky, 1974> paradigm, and Marcus' Wait-and-See Parser <Marcus, 1974>.

If we were to apply Minsky's approach to our P.A. recognition problem, it seems natural to identify the descriptive models of the our knowledge base as the "frames" of Minsky's theory. The descriptive model is thus "invoked" when its clues or "indentifying features" (or IMP'S in Winograd's interpretation), are satisfied by features in the object program. This invocation of a descriptive model corresponds to making the hypothesis. The system then tries to verify that the

hypothesis (frame, model) does in fact fit well. During this phase, the control flow in strongly top-down, i.e. the frame expects certain features to be present a priori, and recognition becomes a case of trying to actually find them in the object. If the hypothesis turns out not to fit, it is abandoned, and the "jump" is made to another one based on the bugs in the current model, and the new information gathered.

Other aspects of recognizing LISP programs have a more bottom-up nature. For instance, there seems to be a need in the P.A. system for templates, (or what Marcus calls "groupers") to recognize such standard structural units as DO-loops, GOTO-loops, etc. This is analgous to Marcus' use of groupers to conglomerate noun phrases or verb groups in parsing natural language. These groups are then passed up to the next level of recognition, where they are fitted into more abstract descriptions. In the P.A. this next level of abstraction would be the plans, corresponding roughly to sentences in the natural language situation.

In any case, these remarks suggest that we need to do more research in the area of control structures for our P.A. To this end, let us give now the following list of relevant issues that are indicated by our investigation thus far:

(1) Top-down and bottom-up.

The control structure clearly will have to support information flow in both directions.

(2) Multiple sources of information.

As we have seen, the P.A. needs to pick up clues from the code and annotation, and guidance from the knowledge base. It should take advantage of the most useful information for each recognition subproblem, regardless of source. This implies a sophisticated arbitration mechanism between information sources.

(3) Incomplete knowledge.

Almost certainly, there will be situations in which the P.A. does not know enough to be able to perform in the manner desired by the user. In such cases, partially useful behaviour, rather than complete failure, should result. For example, the P.A. should be able to take advice and assimilate new information to recover from incomplete knowledge.

(4) Contradictions.

In a similar anthropomorphic vein, the P.A.'s control structure should be able to tolerate contradictory information, both in the knowledge base, and as inputs to the recognition process.

(5) Order of recognition.

In cases where the order in which component parts of a program structure are recognized is intrinsically arbitrary, the control structure should not be capriciously sensitive to the order.

3.4 Advice Taking and Assimilation of New Information

Situations in which the P.A. has incomplete knowledge can be divided into two classes, with respectively appropriate recovery behaviours. The first case is typically when the P.A. is trying to recognize a program it has never seen before, but the programmer has provided insufficient annotation. In this case, the appropriate behaviour is for the P.A to initiate an advice taking interaction with the user. This will be particularly effective because the P.A. will be able to ask for help in an intelligent fashion, i.e. by asking very pointed questions. For example, in the summation program example, the P.A. might ask

```
What implementation have you chosen -

        (a) GOTO-loop?
        (b) DO-loop?
        (c) Recursion?
```

or

```
Where do you bump the counter?
```

or

```
What variable is serving to accumulate the sum?
```

In all these cases, the P.A. has figured things out to a certain point, and uses this partial knowledge to compose a specific and pertinent question to the user .

The second class of incomplete knowlege is more profound. Suppose a user is using a new data structure or programming technique which is not in the P.A.'s knowlege base. The P.A. needs to have mechanisms to assimilate this new information. In the simpler case of a new data structure, one way to do this would be to ask the programmer to explain the new construct to the apprentice. Given that the P.A. knew some things about data structures in general (e.g. they have associated composer and decomposer functions), this interaction could be facilitated by the P.A. prompting the programmer for the relevant information. An elegant and very powerful solution to the assimilation problem, which might be better for learning about new programming techniques,is suggested by the fact that the model built to represent a new user program has the same forms as the permanent knowlege in the knowlege base. What is suggested is a general technique for taking a specific program model, variable-izing it appropriately (herein lies the difficult problem), so it can then be inserted in the knowledge base as a permanent descriptive model. We do not suggest we have a way of doing these things, but we do feel that the system we are developing lends itself well to research in this direction.

3.5 Relation to Natural Language Understanding

There are some interesting parallels that can be drawn between understanding a program you have never seen before, and understanding sentences in natural language. In both cases, a key component in the understanding system is the background knowledge base, which establishes a context for understanding the semantics of the particular utterance in question. The huge problem in natural language understanding research is that if you try to advance beyond conversations in

toy domains like the blocks world, this background knowledge quickly amounts to having a common-sense model of the whole world of human existence. Unfortunately, building such a representation of the world is exactly the central unsolved research project of the entire A.I. community. However, in the case of building a programming apprentice, we believe our research will confirm that the knowledge base for understanding programs is manageably small and well-defined.

The parallel goes deeper than this also. Consider the roles of syntax and semantics in understanding the two kinds of utterances. In the case of natural language, there is a problem with making the semantics sufficiently strong to guide the recognition process, but fortunately (and perhaps not coincidentally) the syntax of natural language carries a lot of information. Thus , a lot of meaningful processing can be done, especially at the low level (such as aggregating noun phrases and verb phrases), without much real understanding of what the sentence means. In the case of LISP programs, however, the basic syntax is so simple and regular that it carries almost no information at all. Programs are understood only by invoking the precisely defined semantics of the LISP "lexicon" (i.e. the basic LISP functions), and the strong models of the background knowledge.

Thus, to summarize, our research is very much *compltmentary* to current natural language understanding research. In both cases, as suggested previously, the control structure issues are very similar: top-down and bottom-up, multiple sources of information, etc. However, we contrast nicely on the relative predominance in the recognition process of syntax vs. semantics . In the case of our research on program understanding, we are able explore the role of background semantics to a much greater extent.

BIBLIOGRAPHY

Balzer, (1973) "Automatic Programming", Institute Technical Memo, University of Southern California / Information Sciences Institute, Los Angeles, Cal.

Burstall, R. M. (1969) "Proving Properties of Programs by Structural Induction", Comput. Jl. vol. 12, pp. 4-8

Burstall, R. M. (1972) "Some Techniques for Proving Properties of Programs Which Alter Data Structures", Machine Intelligence 7, Edinburgh University Press.

Dahl, O.j., Dijkstra, E., And Hoare, C. A. R. (1972) Structured Programming, Academic Press, 1972.

Deutsch, Peter (1973) "An Interactive Program Verifier", Xerox PARC Report CSL-73-1. Palo Alto, Ca.

Floyd, R. W. (1967) "Assigning Meaning to Programs", Mathematical Aspects of Computer Science. J.T. Schwartz (ed.) vol. 19, Am. Math. Soc. pp. 19-32. Providence Rhode Island.

Goldstein, Ira (1974) "Understanding Simple Picture Programs" PhD. Thesis, M.I.T. A.I. Lab. Technical Report 294.

Hewitt, Carl.(1971) "Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", M.i.t. Ai Memo No. 251.

Hewitt, C., Bishop, P., And Steiger, R. (1973) "A Universal Modular Actor Formalism for Artificial Intelligence", Proceedings of IJCAI-73, Stanford California.

Marcus, Mitchell (1974) "Wait-And-See Strategies For Parsing Natural Language" M.I.T. A.I. Working Paper 75.

McDermott, D. V., and Sussman, G.J.(1972) "The CONNIVER Reference Manual", M.I.T. Artificial Intelligence Laboratory, Memo 259.

Minsky, Marvin (1974) "A Framework for Representing Knowledge" M.I.T. A.I. Memo 306.

Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. (1972) "QA4: A Procedure Calculus for Intuitive Reasoning", Stanford Research Institute, Artificial Intelligence Center, Technical Note 73, Menlo Park, Ca.

Ruth, Gregory (1973) "Analysis of Algorithm Implementations" M.i.t. Phd. Thesis, Project Mac Technical Report 130.