MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Working Paper 147 May 1977

PLAN RECOGNITION IN A PROGRAMMER'S APPRENTICE

(Ph.D. Thesis Proposal)

by

Charles Rich

Brief Statement of the Problem:

Stated most generally, the proposed research is concerned with understanding and representing the teleological structure of engineered devices. More specifically, I propose to study the teleological structure of computer programs written in LISP which perform a wide range of non-numerical computations. The major theoretical goal of the research is to further develop a formal representation for teleological structure, called plans, which will facilitate both the abstract description of particular programs, and the compilation of a library of programming expertise in the domain of non-numerical computation. Adequacy of the theory will be demonstrated by implementing a system (to eventually become part of a LISP Programmer's Apprentice) which will be able to recognize various plans in LISP programs written by human programmers and thereby generate cogent explanations of how the programs work, including the detection of some programming errors.

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 the Office of Naval Research contract N00014-75-C-0643.

Section I. Theoretical Background

The purpose of this section is to define the meaning of the underlined words in the brief statement of the problem, leading to a discussion of the proposed research area in the most general and abstract terms possible. To emphasize that the approach being taken here has very wide applicability, I will in this section give examples from three different areas of engineering: electrical, mechanical, and computer programming. Section II will outline my specific and more limited proposal to study the teleological structure of non-numerical LISP programs and for the implementation of a demonstration system for plan recognition.

Engineered Device

By "engineered device" (Sussman [1] uses the term "intentional artifact") I mean a compound object created out of simpler objects to serve a particular purpose. An electronic circuit, a padlock, and a computer program are three examples of engineered devices.

Teleological Structure

A device or a part of a device can be described in two fundamentally different ways. Some properties of a device are independent of its context of use, i.e. independent of its purpose. These properties constitute what I call the intrinsic (or "physical") structure of the device. For example, the current-voltage plot of an electronic component is an intrinsic description; so is identifying a part of a complex mechanical mechanism as a "rod"; or saying that the function of the LISP expression (CAR X) is to take the left half of the dotted pair which is the value of X.

Particular fields of engineering have evolved specialized language for describing the intrinsic structure of devices in that domain. For example, in electronics a circuit may be described by a schematic diagram; in mechanical engineering the physical structure of a device may be described in terms of the orientations, connections, and degrees of freedom of movement of its sub-parts in 3-space; a computer program may be described in terms of control flow and data flow.

However, intrinsic descriptions are often not the most useful for say, explaining to someone how a device works, or trying to figure out what's wrong with it when it malfunctions. More important in these situations are the attributes of a device that derive from the way it is being used, in other words, its teleology.

Devices with the same intrinsic structure may be used for many different purposes. For example, resistors appear many places in a typical electronic circuit used as to bias transistors, as part of voltage dividers, and so on -- yet each resistor is described intrinsically by the same equation V=IR. Similarly, a rod is intrinsically always a rod, regardless of what role (e.g. connecting rod, supporting rod) it plays in a larger mechanical device. In a typical LISP program the CAR function may used in many different places for very different purposes -- in one part of a program (CAR X) may be extracting the "key" field of record X, while in another place (CAR S) is an access the "top" of stack S.

The main thesis of my research is that deep understanding of an engineered device requires not only a representation of its intrinsic structure but also of the purposes (or roles) of each sub-part in the overall design. The interrelated purposes of the parts of a device form a kind of structure which is fundamentally different from, although obviously related to, the intrinsic structure of the device. I call this deeper kind of structure teleological structure, and use the term plan for a description of teleological structure.

Plans

The idea of a plan can be explained most easily in the context of engineering design. By design I mean a particular class of problem solving wherein the goal is to create a device with a specified function. I will rely on an informal understanding by the reader of what kinds of problem solving fit naturally into the design paradigm. For example, we should agree that solving a cryptarithmetic puzzle is not profitably viewed as a design problem.

I distinguish two major phases in the design process, as shown below. The only justification I can currently give for this factorization is that behavior in the early phase of design appears sufficiently different from that in the later phase for it to be useful to make the distinction. At a later point in the research I hope to provide stronger arguments on this point.

```
SPECIFICATIONS (goal)
      |
      | problem solving
      v
    PLAN
      |
      | implementation
      v
   DEVICE (solution)
```

In this simplified view of design, a plan is the major intermediate representation between the neutral problem statement (i.e. the specifications of a desired function), in which there is no indication of how a solution may be obtained, and the complete detailed solution device.

A plan typically describes a decomposition of the original specifications into smaller, related specifications which constitute a possible solution. Finding a plan is the bulk of what is normally called "problem solving" in the AI literature. It may involve trying several different decompositions to solve the problem, and perhaps debugging them as in Sussman's concept of "problem solving by debugging almost right plans" [2]. Candidate plans in this phase of design may be generated either by applying general methods such as means-end analysis (GPS [3]) or linear decomposition (HACKER [4]), or by retrieval from a library of stored plans (PLANNER [5]).

Given a correct plan, the process of transforming it into an actual device is significantly different in character from finding the plan in the first place. I call this second phase of design implementation. Notice that in this simple theory I have not explicitly provided for the not-uncommon occurrence of having to abandon what was thought to be a correct plan because of insurmountable implementation difficulties. However, the existence of feedback between the two phases does not invalidate the initial distinction between them.

To elaborate, let me give some informal examples of plans. In electronics, a simple design problem might be to design an amplifier with an input resistance of 30K ohms and a voltage gain of 5. One plan (there are others) to achieve such a specification is a cascade of a common-collector and a common-emitter circuit. Similarly, in mechanical engineering there many different basic different plans for achieving the function of a lock -- e.g. combination lock, padlock. Finally, in programming there are many different plans for achieving the basic specifications of an associative retrieval data base -- for example using a hash table or a discrimination net.

Plan Recognition

I can now be more specific about what I mean by recognition in the context of plans and devices. Recognition is very close to what is loosely called "understanding" a device; the term recognition puts extra emphasis on the existence of some prior knowledge. Understanding of a device is usually demonstrated by an ability to explain the relationship between the intrinsic structure of the device and how it performs its function. In my terminology, this means having the plan of the device.

It is important not to confuse plan recognition with analysis of the intrinsic structure of a device. For example, real understanding of an electronic circuit is not demonstrated by simply tracing the schematic as in: "R1 is connected to the base of transistor Q1, whose emitter current flows into the collector of Q2 ... "; nor is understanding of a computer program demonstrated by reading it aloud: "first you set R to 2, then set S to R times R, then if (R - X) is less than EPSILON ...".

In contrast, descriptions such as: "this circuit is a two-stage amplifier; Q1 is the input transistor, R1 is a biasing resistor for Q1, C1 is a coupling capacitor, ..." or "this program calculates square root by successive approximation; R is the current approximation, EPSILON is the allowable error, ..." do demonstrate recognition of both the intrinsic and teleological structure of the respective devices.

One major feature of plan recognition is that is provides an appropriate segmentation of the device, i.e. groupings of subparts into units that can be identified with descriptive concepts in the particular domain of engineering. For example, observing that the components of an amplifier circuit may be conceptually partitioned into two stages is a major step forward in understanding how the circuit works. Similarly given a page-long LISP function, segmenting it into conceptual units, such the set-up, a main step, an interface, and another main step (terminology from Goldstein's thesis [6]) is an important first step towards understanding it in more detail.

A second prominent feature of the teleological descriptions above, as compared to the intrinsic descriptions, is the use of a rich domain vocabulary, such as "biasing resistor", "coupling capacitor", "approximation", and "allowable error" to identify various parts. Thus plan recognition has distinguished between different uses of the same kind of device.

Library of Plans

It has become clear in AI research that intelligent problem solving includes in large part an ability to match problem statements against known solution structures. This poses two major questions:

(1) How is a large body of solutions stored and accessed ?
(2) What does a solution structure look like ?

For example, in the theory of PLANNER, storage and retrieval of solutions is achieved by attaching a pattern (in the literal syntactic sense) to each solution; the knowledge base of solutions is then conceptually just a list of solutions, each having a descriptive pattern. Retrieval occurs by matching the problem statement against each pattern and returning a list of those solutions whose pattern matches succeed. The matching and retrieval process can be speeded up by data base techniques such as hashing, but this is not a conceptually important feature. Thus a solution library in the PLANNER theory has no internal structure such as abstraction or generalization hierarchies. Furthermore, in PLANNER the solution itself is a procedure to be executed by the interpreter, i.e. a "black box" with no accessible internal structure (although some structure can be recovered from the solution by the use of a special interpreter such as in HACKER).

My theory of a plan library will differ from PLANNER on both major points. First of all I propose that the structures stored in a solution library should not be totally implemented (or perhaps trivially variable-ized) versions of previous solutions, but rather the plans of previous solutions. Storing plans allows the essential structure of previous solutions to be transferred to current problems, while allowing flexibility in re-implementing the plan to fit a new problem. A further advantage of storing solutions in this more declarative way is that it should allow more diversity of use. In a large and and multiple function system like a programmer's apprentice, the plan library will be shared by many sub-systems, such as design, analysis, and explanation.

As a second major difference, I propose to organize a plan library around abstraction hierarchies. One argument for this organization is purely on the aesthetic grounds that a good theory of knowledge representation should capture the generalizations as well as the specifics of a domain. Furthermore, imposing a hierarchical structure on the plan library allows a retrieval strategy that is suggestive of what is called "recognition" in the context of vision research. For example, in the current theory of Marr and Nishihara [7], the library of possible 3-d models to be recognized in an image is organized in a hierarchy in which the topmost nodes are no particular real objects, but rather very abstract models.

Retrieval from this sort of library has a very different character from the pattern-directed invocation of PLANNER. In a hierarchical library the features of the problem that are expected to be most stable (i.e. least likely to be revised in the light of more detailed analysis) are expressed at the top level nodes. In the Marr and Nishihara theory, for example, the very abstract models describe only the most basic 3-d features of an image, such as number and relative orientation of the main axes. Initial recognition of certain gross features of the problem (e.g. the major axes of a figure) immediately causes retrieval of an appropriate general plan from the library. As the description of a problem becomes more refined, so does the plan for its solution.

Of course, it is possible go down "garden paths" in this approach. Looking at the details may sometimes force one to go back up to the top level and switch to a different major plan type, but this does not invalidate the basic theory any more than the existence of optical illusions necessarily invalidates a given theory of normal visual recognition. In fact, if the observed instances of garden path can be predicted by the theory, they then become strong confirmatory evidence.

Section II. Specific Implementation Proposal

In order to study teleological structure, you have to study the teleological structure of something (an adaptation of Seymour Papert's advice that "in order to think about thinking, you have to think about thinking about something"). I have chosen to study the domain of non-numerical LISP programs because a theory of plans of this type will make possible an important practical application, namely the construction of a LISP programmer's apprentice (see Appendix B).

Synthesis vs. Analysis

In a programmer's apprentice the plan library will be shared by many different active sub-systems. These uses may be categorized into two major modes: synthesis and analysis. Synthesis entails using the library of plans for automated design, i.e. given only a specification of desired behavior, construct the plan of a device which satisfies it. McDermott [8] has worked on this design problem in the domain of simple electronic circuits. The other major use is to understand designs already constructed by human engineers. Understanding (i.e. plan recognition) is demonstrated by the system's ability to give teleological explanations, and to detect (though not necessarily be able to correct) certain kinds of errors.

Given the existing time constraints, I have chosen only one of these modes, namely analysis, as the basic demonstration of adequacy of the proposed theory. This is the most sensible choice for two reasons. First of all, I believe a priori (and of course, only investigation will reveal if I am right) that progress towards my goal of developing an adequate plan library for a large class of of non-numerical programs will be much greater pursuing the analysis route, as compared to working on an automatic programmer for the domain. Moreover, the analysis route allows an evolutionary approach to the project. Since I am interested in understanding the structure of useful programs written by proficient programmers, rather than the beginning exercises of novice programmers, I expect there to be parts of such programs whose understanding will be beyond the current capabilities of the system. Given this realistic expectation of only partial adequacy of the theory in many situations, an important question is: which is more useful, partial analysis of a program or a partial synthesis?

An analysis can be partial in several different dimensions. The simplest kind of incompleteness is on a part-by-part basis, i.e. the system may not be able to recognize the function of a particular part in the larger plan. More subtle kinds of deficient analysis occurs when the system is ignorant of some of the domain facts and rules upon which the operation of a program is based. Nonethless, many useful explanations and bug warnings may be generated on the basis of the parts of the program that are understood. This potential for useful partial performance is the main advantage of analysis over synthesis as a research strategy in this work.

In contrast, consider what "partial design" would mean. One possible meaning is design down to some level of detail which is still more abstract than an actual program. In such a situation the human programmer would be responsible for implementing the details of the design and actually coding the program while preserving the correctness of the original design. The alternative notion of partial design, i.e. design of a device that only partially works, we can surely agree is extremely unappealing.

A Demonstration System

As a demonstration of my theory of plans and plan recognition, I propose to implement a rudimentary plan recognition system (hereafter referred to as "the system"). The block diagram of the system is as follows:

$$\begin{tikzcd}
 & & \text{explanations} \\
\text{Commented LISP program} \arrow[r] & \text{RECOGNITION MODULE} \arrow[ur] \arrow[dr] & \\
 & & \text{error warnings} \\
 & \text{PLAN LIBRARY} \arrow[uu] &
\end{tikzcd}$$

In this diagram the two central blocks, "recognition module" and "plan library" are the major objects of research. The rest of this section will define more precisely the exact form of the inputs ("commented LISP program") and outputs ("explanations" and "error warnings").

It is important to note that this proposed system has two major intentions: first to be a convincing demonstration of the adequacy and feasibility of the approach to program understanding which I are putting forward, and second, to be the initial implementation of parts of a more comprehensive programmer's apprentice system. The relationship between this system and the other major sub-system of the P.A. being worked on by H. Shrobe is discussed in Section IV.

The Domain of Input Programs

The programming domain in which the system will have inherent expertise can be described generally as non-numerical (i.e. symbolic) computations of moderate complexity. Appendix A is a typical example. Notice that this program includes instances of many of the basic data structures and algorithms in this domain of programming:

Basic Data Structures	Basic Algorithms
arrays	search
lists	enumeration
ordered lists	insertion
hash tables	deletion
trees	intersection
	union
	membership

This list suggests the minimum plan library that a useful programmer's apprentice should have for non-numerical programming. The list is not yet complete; I expect that in the course of actually constructing the plan library, its generalization hierarchy will naturally point out missing plans.

Many different programs can be written using these basic building blocks. It could be said that the domain of input programs is defined inductively as those programs whose plans are found in the basic plan library, or whose plans are compositions (in an appropriate sense) of plans in the basic plan library. The exact nature of the plan composition operators cannot be specified at this time, and is one of the interesting results I hope to obtain from the proposed research. However it is clear that overlapping and intermingling plans will be an allowable and important form of composition.

Furthermore, because the system will be able to perform partial recognition as discussed above, delimiting the exact domain of allowable input programs is less crucial than it would otherwise be.

Programmer-Supplied Commentary

Programmer-supplied commentary wil play two major roles in the recognition system, which I term <u>definitional</u> and <u>annotative</u>.

The need for definitional comments is due to the fact that a general-purpose programmer's apprentice obviously cannot have inherent knowledge of all applications that could be implemented using the data structures and algorithms in its basic library. Part of a programmer's commentary will therefore be concerned with defining the vocabulary of objects, relations, and operations in his application domain that are relevant to understanding the rest of his program. The first twenty lines of the program listing in Appendix A are an example of definitional commentary.

Sometimes these definitions constitute only the programmer's idiosyncratic terminology for concepts which the apprentice already possesses. For example, line 007 on page one of Appendix A defines the term "form" to mean an s-expression which is a member of the table. The basic LISP apprentice will have inherent knowledge of s-expressions and the general concept of membership relations. The importance of such a comment is to allow the system to understand the programmer's use of the term "form" later, and to use this term itself in explanations and error messages it generates for the programmer.

Some definitional comments introduce totally new concepts, as for example lines 011 through 014 which define a "bucket" in terms of CAR , CDR and lists. Thus another role of definitional commentary is to record the programmer's decisions of how to implement his domain objects in terms of the "primitives" (at whatever level) available. Notice I am assuming in this example only a very basic apprentice that knows about arrays, lists, and the other structures out of which hash tables are built, but for which the hash table data base itself is an application program.

In contrast with definitional comments, which tend to be grouped at the beginning of a program or between function definitions, annotative comments are interspersed throughout the body of code and usually pertain specifically to the s-expression(s) either immediately preceding or immediately following. These comments will help the recognition system greatly in making the detailed correspondence between the intrinsic structure of a program and its teleological structure. For example line 055 informs the system that the purpose of the expression (OR (TABLE (HASH KEY)) (RETURN NIL)) is to test for the empty bucket. Similarly, the comment on line 066 will aid plan recognition by informing the system that the LISP variable INT is being used locally to hold the accumulated list of forms in the intersection of the buckets.

For the demonstration system I do not propose to deal with the problems of accepting programmer-supplied commentary in unconstrained natural language. However, I am very much interested in developing a theory of what the content of such commentary should be in the context of plan recognition. To facilitate this investigation, I intend to develop a very simple English-like formal language in which program commentary is to be written. Because the design of this language will incorporate a theory of what needs to expressed in this situation, I hope that programmers will find it quite convenient to use, despite its being highly constrained syntactically.

Output from the System

Output from the demonstration system will be of two forms. First of all, in response to specific queries from the user, the system will be able to generate teleological explanations of how the given program works. Error warnings, the second form of output, occur on the initiative of the system, drawing the programmer's attention to errors or potential errors in his program that have been detected while trying to recognize the plan.

The only criterion for judging the output of the demonstration system is that it should contain correct and adequate information for the situation. I will not deal with how to abbreviate these messages appropriately (as would be required for a usable system) or how to state them in fluent English. As a convenience, however, I do intend to implement a very simple English-like formal language for requesting explanations and in which the text of explanations and error messages can be generated by simple template-filling. Furthermore, it might be particularly useful to employ the same language (or an extension) for both query and output messages as is used for programmer-supplied commentary. This would allow several interesting possibilities, such as the system being able to re-read the explanations it generated as if they were programmer-supplied commentary.

Explanations

Since the main purpose the proposed implementation is to demonstrate an ability to recognize the correspondence between the intrinsic structure of a given LISP program and its teleological structure, the only kind of explanations generated will be in answer to questions about this relationship. In effect, I am assuming that the user understands the plan of the program, and his need is in understanding how the plan is realized in details of the code. Obviously there are many other kinds of explanation that would be useful, such as a discussion of design decisions, explanation of program traces, and so on. These modes of explanation are envisioned for the complete programmer's apprentice.

A typical question regarding the example program of Appendix A might be "what is the purpose of line 059 on page one", to which the system should answer something like "to sort the buckets by length". This is an example of a question where the user points out a component (ie. s-expression) in the intrinsic structure of the device, and wants to know its role in the teleological structure. Alternatively, a user could describe some action in the plan and enquire where in the program it is achieved. To illustrate this form of question, simply invert the example above and have the user ask "where are the buckets sorted by length", to which the system should respond "in line 059 on page one".

The other basic form of question the system will be able to answer concerns data flow and use of variables in the LISP program. For example, a user might enquire "what is L in the s-expression (NULL L) on line 069 ?", to which the system should respond something like "the list of buckets which have not yet been intersected". Notice that in this example no comment has been provided by the programmer which explicitly answers this question. The system will discover this fact as part of recognizing the plan of the LOOKUP function.

Error Detection

I distinguish three levels of programming errors: design errors, implementation errors, and coding errors. The most serious kind of errors are those in the fundamental design of the program, that is, incorrect plans. The recognition system will not be able to detect such errors. It will assume that plans in the library have already been verified. New plans presented by the programmer will have to be accepted at face value. In the eventual programmer's apprentice however, a general purpose plan verification facility, such as Shrobe's system (see Section IV), will be provided.

Next in degree of seriousness are implementation errors. An implementation error has occurred when there is an inconsistency between the intrinsic structure of a program and its plan. Examples of this are when the prerequisite of an action does not precede it, or when the actual data flow in a program does not correspond to the data flow required by the plan. I also categorize as implementation errors inconsistencies between implementation decisions in different parts of the program, as for example if a programmer in one place used CAR to extract the "key" field of a given type of object, and in another place used CADR. Implementation errors are the major kind of error the recognition system will be able to detect.

Coding errors are the most superficial kind of error. In this category I include misspellings, syntactic errors such as unbalanced parentheses, and misuse of the language primitives such as using RETURN inside of PROGN. The demonstration system will not concentrate on the detection of this kind of error, since effective syntactic techniques for this have already been developed in other systems, most notably INTERLISP [17].

Section III. Current State of the Theory

In this section I comment briefly on the current state of the theory in four main areas:

(1) Description of the intrinsic structure of LISP programs.
(2) Description of teleological structure.
(3) Organization of a plan library.
(4) Plan recognition.

In this section "we" refers to the joint authors of TR-354 (Appendix B), H. Shrobe, and the present author, C. Rich. For more detailed discussion of these topics and many others related to the construction of a LISP programmer's apprentice, please refer to Appendix B.

Our overall conceptual framework for understanding programs is the view of data objects flowing between modules that operate upon them (including operations which have side effects). This is obviously not the only possible way to think about programs (for a totally different view see [18]), but it is the one that most practicing LISP programmers find natural, and this was the crucial criterion for our choice.

The basic unit of description for both intrinsic structure and teleological structure is called a segment. Data objects flow into a segment and new or side-effected data objects flow out. Segments in a plan also have specifications. The specifications of a segment are a formal statement of input expectations (conditions on or relationships between input objects that are expected to hold prior to the execution of the segment), and output assertions (conditions that will hold of the output objects immediately after execution).

A plan is represented as a hierarchical network of segments with various kinds of links between them. The intrinsic structure of a program is expressed by data flow links, in which the output object of one segment becomes the input object of another segment, and control flow links, which specify order of execution. A program's teleological structure is expressed by purpose links, which relate the specifications of segments to one another. The two most basic types of purpose link are the prerequisite link, in which the input expectations of one segment depend on the output assertions of another, and the achieve link, in which the output assertions of a sub-segment are linked to the output assertions of the segment of which it is a part. Thus the end result of recognition is a single unified description in which both the intrinsic and teleological structure of the program is represented.

Plans are intended to capture the procedural aspects of programming. We have also developed a formalism for describing the structure of data, based on the natural concepts of a part (e.g. a dotted pair has two parts, CAR and CDR),a generic part (e.g. an array has generic parts called items), and relationships between parts (e.g. the count part of a bucket is equal to the number of forms in the contents).

The detailed organization of a plan library is still to be developed. Thus far we have identified two main techniques by which we can obtain generative power from our plan representation and thereby eliminate needless redundancy in the library.

The first technique is specialization. The plan for searching linear data structures should only be expressed once in the library as a single very abstract description. The system should then be able to specialize this plan according to the descriptions of a variety of particular linear data object types, such as arrays, lists, or user-defined object types.

Another kind of redundancy to be eliminated are the many minor variations of a plan due to grouping the sub-segments slightly differently. I hope to capture this kind of variation in the form of general transformations that can be applied to many plans.

As for the gross hierarchical structure of the library, the topmost distinctions in the hierarchy will be on the basis of the loop and recursion structure of the plans, e.g. iterative loops, single recursions, double recursions, and so on. Loops, for example, can then be sub-categorized into searching loops, counting loops, approximation loops, and so on.

On the topic of the recognition process itself, several main themes have emerged thus far:

(i) propagation of discovered information
(ii) bottom-up grouping of segments
(iii) goal-directed grouping and plan transformation
(iv) hypothesis generation and testing

Also, some capability for reasoning about program behavior will be required for recognition. In the eventual programmer's apprentice, recognition would have the full resources of a system like Shrobe's available to it, but for the currently proposed system, some much more limited reasoning module will have to be designed.

Section IV. Relation to Other Work

The reader is referred to Chapter Six of Appendix B (TR-354) for a more lengthy discussion of the other work leading up to and concurrent with the LISP programmer's apprentice project as a whole. In this section I will elaborate only upon the relationship of the proposed plan recognition system and plan library to other work.

Ruth's Analysis of Algorithm Implementations

The previous work most comparable with the recognition system is Greg Ruth's Ph.D. thesis at MIT [9]. Ruth constructed a system which succesfully analyzed correct and near-correct PL/I programs from an introductory programming class, giving specific comments about the nature of the errors detected in the incorrect programs. In Ruth's system, the class of expected programs for a given exercise is represented as a formal grammar augmented with global switches which control conditional expansions. This grammar is then used in a combination of top-down, bottom-up, and heuristic-based parsing in order to recognize particular programs.

Ruth's work has two fundamental shortcomings. First of all, his analysis of programs does not include any form of specification -- i.e. there is no explicit statement of what a program is attempting to achieve, or what any of the subparts do individually. Thus Ruth's analysis never really captures the teleological structure of a program.

Furthermore, even given input-output specifications for non-terminal nodes, parse trees derived from a formal grammar are inadequate for representing many important forms of teleological structure. Ruth's explanation of the purpose of a given action in a program is limited to an upward trace through the non-terminal nodes dominating the action in the program's parse tree. Tree structure is adequate to represent goal-subgoal relationships (i.e. ACHIEVE links), but does not make a crucial distinction between steps that just happen to precede each other and actual prerequisite constraints. Furthermore, the restriction to tree-structured analysis precludes one program action from having two different purposes; or put another way, it precludes overlapping (or interleaving) the actions which implement distinct modules at a higher level of description.

To summarize, Ruth's analysis of programs gives an adequate description of teleology only for totally hierarchical programs. For programs in which their are many dependencies between modules, Ruth's formal grammar representation is inappropriate. The system I am proposing will be able to analyze and detect errors in at least all the programs that Ruth's does (given their translations from PL/I to LISP). Furthermore, since my system will represent more of the teleological structure of these programs, I will be able to provide better explanations and error messages.

Miller & Goldstein's Planning Grammars

Miller and Goldstein [10] at MIT are also working on a codification of programming knowledge that can be used for analysis and correction of student programs. Like Ruth, they represent programming knowledge as a formal grammar that will generate all members of the class of programs they are prepared to recognize. Thus heuristic parsing techniques are also the essence of their program recognition methodology.

Miller and Goldstein's grammar has several improvements over Ruth's which mitigate my major objections. First of all, Miller and Goldstein have added specifications (called pre- and post-models) to the nodes of their grammar. This allows them to install prerequisite links between the post-models of program steps and the pre-models of other steps at the same level in the tree that depend upon them. Furthermore, in Miller and Goldstein's augmented transition network implementation of their grammar, they are prepared to allow a single program action to be parsed into more than one higher level node.

The main difference between Miller and Goldstein's grammar representation of programming knowledge and my plan library is one of emphasis. Their grammar emphasizes the intrinsic structure of programs, with the teleological relationships such as prerequisite links added as annotation of the derivation tree. In my view the most fundamental program representation is a network of purpose links between specification segments, with the intrinsic structure, such as execution ordering, being a feature of a particular user's program discovered as part of the recognition process.

Green & Barstow's Rules for the Automatic Synthesis of Programs

The largest existing machine-usable codification of programming knowledge for the domain of non-numerical LISP programs has been compiled by Green and Barstow [11] as part of the PSI automatic programming project at Stanford University. Their codification consists of rewrite rules that progressively and hierarchically refine the description of a desired program in a very high level language into a correct implementation in LISP. Furthermore, since Green and Barstow are concerned with automatic synthesis of efficient programs, the rules in their generative grammar are annotated with pragmatic information which the PSI system uses [19] to select efficient implementations from among all possible correct implementations.

As a representation of programming knowledge for use in a programmer's apprentice, the PSI rules suffer from the same two shortcomings as were described above for Ruth. Having been developed primarily for synthesis rather than analysis and explanation, the PSI rules do not provide any representation of the teleological structure of the resulting program other than a history of the rule applications. Furthermore, the existing rule library includes only rules for hierarchical refinement. Program transformations which overlap module boundaries are not included in Green and Barstow's current theory.

In comparison, my plan library will include knowledge all of the same basic data structure and programming techniques as have been codified by Green and Barstow (with the exception of specific expertise on sorting methods, which has been a major development focus for the PSI system, but will not be for my system.) However, as described earlier, my library will be very different in both its gross organization (i.e. hierarchical, rather than production system-like) and in its detailed representation of programming algorithms (i.e. using plans with explicit teleological dependencies expressed, rather than refinement rules).

Shrobe's Plan Verification System

Concurrent with my research into plan recognition, Howard Shrobe also at MIT, will be further developing [12] the deductive system described in Chapter Three of Appendix B (TR-354). His work and my own are unified by our common use of the plan representations described in TR-354. The primary function of Shrobe's system will be to prove that a plan satisfies its overall specifications, given only the input-output specifications of each sub-segment and the data flow between them. As a by-product of this plan verification, Shrobe's system will also have discovered and recorded in the plan (as purpose links), all the dependencies between segments that are required to support its correctness.

As a demonstration of the capabilities of his system, Shrobe has chosen an interactive design scenario. In this application, a programmer first specifies his programming task to the system at a fairly abstract level. He then puts forward his plan to achieve the given specifications, using specified sub-segments linked together by appropriate data flow. Shrobe's system attempts to verify the programmer's plan, and if it is found faulty provides him with a description of where the proof breaks down in terms that are understandable and helpful towards his adjusting the plan to make it correct. When the plan is correct, it is stored away in the plan library, having been annotated with a record of the proof in the form of purpose links. Shrobe's system will not deal with actual LISP code.

Shrobe's work and my own are coordinated in several ways that lead towards the future realization of a more complete programmer's apprentice. For example, Shrobe's interactive design system could be interfaced to my recognition system in a simple cascade arrangement, whereby a programmer would first use Shrobe's system to develop a correct plan, and then move on to actually coding the program in LISP, with the recognition system checking for consistency with the verified plan.

The two systems will also share the same plan library. Although for the present the organization of the plan library will be dictated by the needs of plan recognition rather than verification, I strongly suspect that the same hierarchical organization is most appropriate for both tasks. My research will lead to filling the plan library with the pre-compiled, pre-verified, basic building block plans of non-numerical computation. Shrobe's system will be able to add to this basic library additional, more specific plans designed by particular users.

Other Concurrent Research

Other investigations into the teleological structure of engineered devices are currently being conducted at MIT by Waters, DeKleer, and Freiling, and have recently been completed by McDermott and Brown. Their related work is most clearly distinguished from this proposal by the differing domains of study.

Waters [13] is working on a system for analyzing mathematical FORTRAN programs similar to those in the IBM Scientific Subroutine Package. Although there is obviously some overlap between the plans used in numerical and symbolic computations, I expect that in the main Waters will be studying very different kinds of plans than myself.

McDermott, Brown and DeKleer have all three been working in the domain of discrete analog electronic circuits. McDermott [8] has been most concerned with the issues of efficiency and extensibility in compiling a large library of plans to be used in the design of electronic circuits. Brown's thesis [14] describes a system for localizing failures in electronic circuits by making use of teleological descriptions of the circuits. DeKleer [15] has proposed a system which will be able to recognize circuit plans in annotated schematic diagrams, much the same as I intend to recognize plans in annotated program listings.

Finally, Freiling [16] has been working on understanding the teleological structure of mechanical systems.

Appendix A. An Example LISP Program

(see program listing attached)

Appendix B. Initial Report on a LISP Programmer's Apprentice

(see MIT AI Lab TR-354, December 1976, by C. Rich and H.E. Shrobe attached)

References.

[1] Sussman, G.J. "The Engineering Problem Solving Project". A Research Proposal Submitted to the National Science Foundation. MIT AI Lab. July, 1977.

[2] Sussman, G.J. "The Virtuous Nature of Bugs". *Proc. AISB Summer Conference*, U. of Sussex. July, 1974.

[3] Newell, A. and Simon, H. "GPS, A Program That Simulatest Human Thought" in *Computers and Thought*, Feigenbaum and Feldman (eds). McGraw-Hill. 1963.

[4] Sussman, G.J. *A Computational Model of Skill Acquisition.* MIT AI Lab TR-297. September, 1974.

[5] Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot". *IJCAI-69*, Washington D.C. May, 1969.

[6] Goldstein, I.P. *Understanding Simple Picture Programs.* MIT AI Lab TR-294. September, 1974.

[7] Marr, D. and Nishihara, H.K. "Spatial Disposition of Axes in a Generalized Cylinder Representation of Objects That Do Not Encopass the Viewer". MIT AI Lab Memo 341. December, 1975.

[8] McDermott, D.V. *Flexibility and Efficiency in a Computer Program for Designing Circuits.* MIT AI Lab TR-402. (in draft).

[9] Ruth, G.R. *Analysis of Algorithm Implementations.* MIT Project MAC TR-130. May, 1974.

[10] Miller, M.L. and Goldstein, I.P. "Overview of a Linguistic Theory of Design". MIT AI Lab Memo 383. December, 1976.

[11] Green, G.C. and Barstow, D.R. "Some Rules for the Automatic Synthesis of Programs". *IJCAI-4*, Tbilisi, USSR. September, 1975.

[12] Shrobe, H.E. Ph.D. Thesis Proposal forthcoming. MIT AI Lab.

[13] Waters, R.C. "A System for Understanding Mathematical FORTRAN Programs". MIT AI Lab Memo 368. August, 1976.

[14] Brown, A. *Qualitative Knowledge, Causal Reasoning, and the Localization of Failures.* MIT AI Lab TR-362. (in draft).

[15] DeKleer, J. "A Theory of Plans for Electronic Circuits". MIT AI Lab Working Paper 144. April 1977.

[16] Freiling, M. *The Use of a Hierarchical Representation in the Understanding of Mechanical Systems.* Ph.D. thesis forthcoming. MIT AI Lab.

[17] Teitelman, W. *INTERLISP Reference Manual.* XEROX Palo Alto Research Center publication. 1974.

[18] Hewitt C., Bishop P. and Steiger, R. "A Universal Modular Actor Formalism for Artificial Intelligence". *IJCAI-3*, Stanford, Calif. August, 1973.

[19] Kant, E. "The Selection of Efficient Implementations for A High Level Language". Stanford AI Lab. (draft).

```
;;; This is an implementation of a hash table data base similar
;;; to the one used in the original implementation of CONNIVER.
;;;
;;; The hash "table" is an array, called TABLE, of size TABLE-SIZE.
;;;
;;; Members in the table are s-expressions called "forms".
;;;
;;; The table is made up of "buckets" which are the slots of the array.
;;;
;;; A bucket has two parts:
;;;     The CDR of a bucket is a list of forms called the "contents".
;;;     The CAR of a bucket is called the "count", and is
;;;      equal to the number of forms in the contents.
;;;
;;; Each form in the table is stored under several "keys".  These keys are
;;; computed by INDEX.  The HASH function computes an index in the
;;; array (i.e. a bucket) for each key.  Forms in the table are
;;; members of each bucket hashed to by the keys for the form.


(DECLARE (SPECIAL TABLE TABLE-SIZE *KEYS))

(SETQ TABLE-SIZE 10.)

(ARRAY TABLE T TABLE-SIZE)      ;initialize table.


(DEFINE INSERT (FORM)
        ;; insert form in data base.
        ;; note: does not check if form already present.
        (MAPC '(LAMBDA (KEY)
                      (BUCKET-INSERT FORM (HASH KEY)))
              (INDEX FORM)))

(DEFINE DELETE (FORM)
        ;; delete form from data base.
        (MAPC '(LAMBDA (KEY)
                      (BUCKET-DELETE FORM (HASH KEY)))
              (INDEX FORM)))


;;; Forms are retrieved from the data base by a "pattern", which
;;;  is just like a form, except that the atom * is a special
;;;  symbol which matches any other atom.



(DEFINE LOOKUP (PATTERN)
        ;; return list of forms in data base which match pattern.
        (PROG (MATCHES ;to accumulate forms that match.
               BKTS) ;list of buckets to be intersected.
              (SETQ BKTS (MAPCAR '(LAMBDA (KEY)
                                        (OR (TABLE (HASH KEY))
                                            ;; special check for empty bucket.
                                            (RETURN NIL)))
                                 (INDEX PATTERN)))
              ;; sort buckets by length to speed up intersection.
              (SETQ BKTS (SORTCAR BKTS '<))
              (MAPC '(LAMBDA (FORM)
                            ;; select only forms that match pattern.
                            (AND (MATCH FORM PATTERN)
                                 (SETQ MATCHES (CONS FORM MATCHES))))
                    ;; intersect buckets to get candidates for match.
                    (DO ((L (CDR BKTS) (CDR L))
                         (INT ;to accumulate intersection.
                          (CDAR BKTS) ;initialize to contents of first bucket.
                          (FAST-INTERSECT INT (CDAR L))))
                        ((OR (NULL L)(NULL INT))
                         ;; done, return intersection.
                         INT)))
              (RETURN MATCHES)))
```

```
(DEFINE INDEX (FORM)
        ;; returns list of keys for form
        (PROG (*KEYS)
              (INDEX1 FORM 1)
              (RETURN *KEYS)))


(DEFINE INDEX1 (FORM POSITION)
        ;; compute keys and return them by
        ;; cons'ing onto global variable *KEYS
        (COND ((ATOM FORM)
               ;; note that atom * is not indexed
               (OR (EQ FORM '*)
                   (SETQ *KEYS (CONS (BUILD-KEY FORM POSITION)
                                     *KEYS))))
              (T
               ;; tree recursion
               (INDEX1 (CAR FORM) (LSH POSITION 1))
               (INDEX1 (CDR FORM)(1+ (LSH POSITION 1))))))


(DEFINE BUILD-KEY (ATOM POSITION)
        ;; make unique key for atom and position
        (+ (MAKNUM ATOM) (LSH POSITION 18.)))



(DEFINE HASH (KEY)
        ;; return index in table for key.
        (ABS (REMAINDER KEY TABLE-SIZE)))


(DEFINE MATCH (FORM PATTERN)
        ;; predicate which returns T is match, NIL otherwise.
        ;; note: no variables are used, only * which matches anything.
        (COND ((ATOM PATTERN)
               (OR (EQ PATTERN '*)
                   (EQ PATTERN FORM)))
              (T ;; tree recursion
               (AND (MATCH (CAR FORM)(CAR PATTERN))
                    (MATCH (CDR FORM)(CDR PATTERN))))))
```

```
;;; The contents of a bucket is ordered by MAKNUM, which allows faster
;;;  computation of the intersection of two buckets.
;;;
;;; MAKNUM is a function which assigns distinct numbers to s-expressions
;;;  which are not EQ.


(DEFINE FAST-INTERSECT (LIST1 LIST2)
        ;; return intersection of the two lists
        ;; which are ordered by maknum
        (PROG (LIST3) ;to accumulate intersection.
           LP (COND ((AND LIST1 LIST2)
                    (COND ((EQ (CAR LIST1)(CAR LIST2))
                           ;;found element in intersection
                           (SETQ LIST3 (CONS (CAR LIST1) LIST3))
                           (SETQ LIST1 (CDR LIST1))
                           (SETQ LIST2 (CDR LIST2)))
                          ((< (MAKNUM (CAR LIST1))(MAKNUM (CAR LIST2)))
                           (SETQ LIST1 (CDR LIST1)))
                          (T ;;note > holds by elimination
                           (SETQ LIST2 (CDR LIST2))))
                    (GO LP))
                   (T
                    ;; one of lists has run out
                    (RETURN LIST3)))))


(DEFINE BUCKET-INSERT (FORM INDEX)
       ;; splice form into indexed bucket in maknum order.
       ;; note: does not check for duplicates.
       (PROG (BKT MAKNUM)
             ;; bucket fetch.
             (SETQ BKT (TABLE INDEX))
             (SETQ M (MAKNUM FORM))
             (COND (BKT
                    ;; search for place in bucket
                    (DO ((B (CDR BKT) (CDR B))  ;B is ordered list.
                         (PREV BKT B))          ;PREV is pointer for RPLACD.
                        ((OR (NULL B)(> (MAKNUM (CAR B)) M))
                         ;; do insertion.
                         (RPLACD PREV (CONS FORM B))))
                    ;; update bucket count.
                    (RPLACA BKT (1+ (CAR BKT))))
                   (T
                    ;; previously unused bucket.
                    ;; build new bucket with one entry.
                    (STORE (TABLE (HASH INDEX))
                           (CONS 1 (LIST FORM)))))))


(DEFINE BUCKET-DELETE (FORM INDEX)
       ;; excise given item from  indexed bucket.
       ;; note:  does delete duplicates.
       (PROG (BKT)
             ;; bucket fetch.
             (SETQ BKT (TABLE INDEX))
             ;; this is ordinary list deletion.
             (DO ((L (CDR BKT) (CDR L)) ;L is a list.
                  (PREV BKT L))         ;PREV is pointer for RPLACD.
                 ((NULL L))
                 (COND ((EQUAL FORM (CAR L))
                        ;; splice out.
                        (RPLACD PREV (CDR L))
                        (SETQ L PREV)
                        ;; decrement bucket count.
                        (RPLACA BKT (1- (CAR BKT))))))))
```