

PROCEDURAL EMBEDDING OF KNOWLEDGE IN PLANNER

Carl Hewitt

Artificial Intelligence Laboratory,
M.I.T., Cambridge, Massachusetts,
U.S.A.

0. Abstract

Since the last IJCAI, the PLANNER problem solving formalism has continued to develop. Our epistemology for the foundations for problem solving has been extended. An overview of the formalism is given from an information processing viewpoint. A simple example is explained using snapshots of the state of the problem solving as the example is worked. Finally, current applications for the formalism are listed.

1. The Structural Foundations of Problem Solving

We would like to develop a foundation for problem solving analogous in some ways to the currently existing foundations for mathematics. Thus we need to analyze the structure of foundations for mathematics. A foundation for mathematics must provide a definitional formalism in which mathematical objects can be defined and their existence proved. For example set theory as a foundation provides that objects must be built out of sets. Then there must be a deductive formalism in which fundamental truths can be stated and the means provided to deduce additional truths from those already established. Current mathematical foundations such as set theory seem quite natural and adequate for the vast body of classical mathematics. The objects and reasoning of most mathematical domains such as analysis and algebra can be easily founded on set theory. The existence of certain astronomically large cardinals poses some problems for set theoretic foundations. However, the problems posed seem to be of practical importance only to certain category theorists. Foundations of mathematics have devoted a great deal of attention to the problems of consistency and completeness. The problem of consistency is important since if the foundations are inconsistent then any formula whatsoever may be deduced thus trivializing the foundations. Semantics for foundations of mathematics are defined model

theoretically in terms of the notion of satisfiability. The problem of completeness is that, for a foundation of mathematics to be intuitively satisfactory all the true formulas should be provable since a foundation of mathematics aims to be a theory of mathematical truth.

Similar fundamental questions must be faced by a foundation for problem solving. However there are some important differences since a foundation of problem solving aims more to be a theory of actions and purposes than a theory of mathematical truth. A foundation for problem solving must specify a goal-oriented formalism in which problems can be stated. Furthermore there must be a formalism for specifying the allowable methods of solution of problems. As part of the definition of the formalisms, the following elements must be defined: the data structure, the control structure, and the primitive procedures. The problem of what are allowable data structures for facts about the world immediately arises. Being a theory of actions, a foundation for problem solving must confront the problem of change: How can account be taken of the changing situation in the world? In order for there to be problem solving, there must be a problem solver. A foundation for problem solving must consider how much knowledge and what kind of knowledge problem solvers can have about themselves. In contrast to the foundation of mathematics, the semantics for a foundation for problem solving should be defined in terms of properties of procedures. We would like to see mathematical investigations on the adequacy of the foundations for problem solving provided by PLANNER. In chapter 8 of the dissertation, we have made the beginnings of one kind of such an investigation.

To be more specific a foundation for problem solving must concern itself with the following complex of topics.

PROCEDURAL EMBEDDING: How can "real world" knowledge be effectively embedded in procedures. What are good ways to express problem solution methods and how can plans for the solution of problems be formulated?

GENERALIZED COMPILATION: What are good methods for transforming high level goal-oriented language into specific efficient algorithms.

VERIFICATION: How can it be verified that a procedure does what is intended.

PROCEDURAL ABSTRACTION: What are good methods for abstracting general procedures from special cases.

One approach to foundations for problem solving requires that there should be two distinct formalisms:

1: A METHODS formalism which specifies the allowable methods of solution

2: A PROBLEM SPECIFICATION formalism in which to pose problems.

The problem solver is expected to figure out how combine its available methods in order to produce a solution which satisfies the problem specification. One of the aims of the above formulation of problem solving is to clearly separate the methods of solution from the problems posed so that it is impossible to "cheat" and give the problem solver the methods for solving the problem along with the statement of the problem. We propose to bridge the chasm between the methods formalism and the problem formalism. Consider more carefully the two extremes in the specification of processing:

A: Explicit processing (e.g. methods) is the ability to specify and control actions down to the finest details.

B: Implicit processing (e.g. problems) is the ability to specify the end result desired and not have to say very much about how it should be achieved.

PLANNER attempts to provide a formalism in which a problem solver can bridge the continuum between explicit and implicit processing. We aim for a maximum of flexibility so that whatever knowledge is available can be incorporated even if it is fragmentary and heuristic.

Our work on PLANNER has been an investigation in PROCEDURAL EPISTEMOLOGY, the study of how knowledge can be embedded in procedures. The THESIS OF PROCEDURAL EMBEDDING is that intellectual structures should be analyzed through their PROCEDURAL ANALOGUES. We will try to show what we mean through examples:

DESCRIPTIONS are procedures which recognize how well some candidate fits the description.

PATTERNS are descriptions which match configurations of data. For example $\langle \text{either } 4 \text{ } \langle \text{atomic} \rangle \rangle$ is a procedure which will recognize something which is either 4 or is atomic.

DATA TYPES are patterns used in declarations of the allowable range and domain of procedures and identifiers. More generally, data types have analogues in the form of procedures which create, destroy, recognize, and transform data.

GRAMMARS: The PROGRAMMAR language of Terry Winograd represents the first step towards one kind of procedural analogue for natural language grammar.

SCHEMATIC DRAWINGS have as their procedural analogue methods for recognizing when particular figures fit within the schemata.

PROOFS correspond to procedures for recognizing and expanding valid chains of deductions. Indeed many proofs can fruitfully be considered to define procedures which are proved to have certain properties.

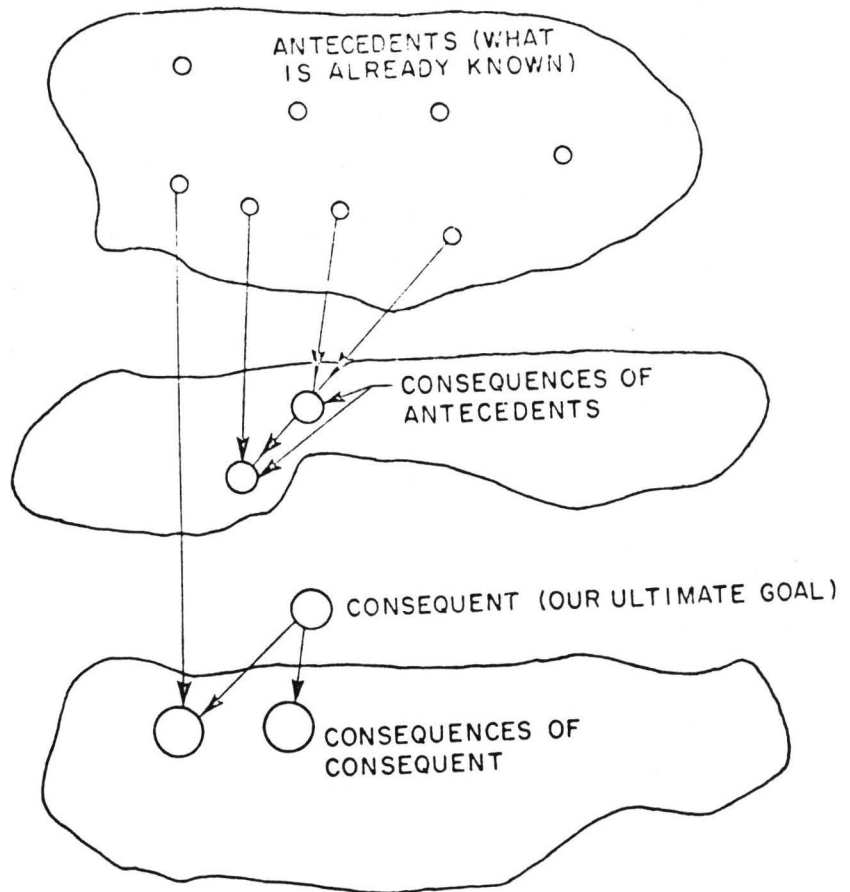
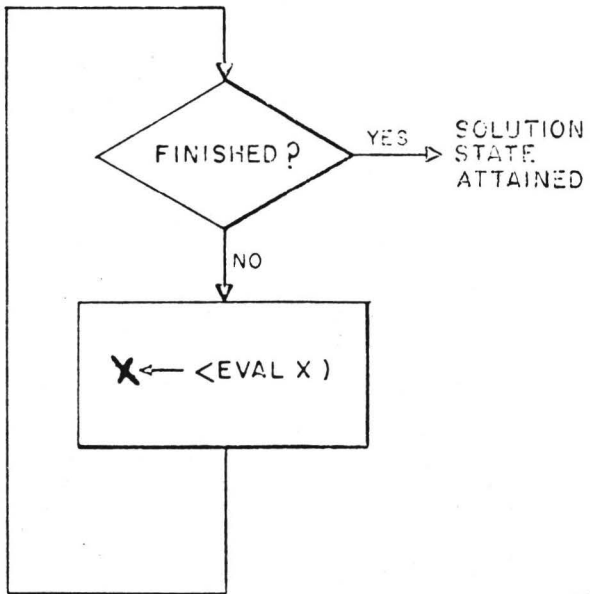
MODELS of PROGRAMS are procedures for defining properties of procedures and attempting to verify these properties. Models of programs can be defined by procedures which state the relations that must hold as control passes through the program.

PLANS are general, goal oriented procedures for attempting to carry out some task.

THEOREMS of the QUANTIFICATIONAL CALCULUS have as their analogues procedures for carrying out the deductions which are justified by the theorems. For example, consider a theorem of the form $(\text{IMPLIES } x \ y)$. One procedural analogue of the theorem is to consider whether x should be made a subgoal in order to try to prove something of the form y . Ira Goldstein has shown that the theorems of elementary plane geometry have very natural procedural analogues.

DRAWINGS: The procedural analogue of a drawing is a procedure for making the drawing.

PROGRESSIVE REFINEMENT



Rather sophisticated display processors have been constructed for making drawings on cathode ray tubes.

RECOMMENDATIONS:

PLANNER has primitives which allow recommendations as to how disparate sections of goal oriented language should be linked together in order to accomplish some particular task.

GOAL TREES are represented by a snapshot of the instantaneous configuration of problem solving processes.

One corollary of the thesis of procedural embedding is that learning entails the learning of the procedures in which the knowledge to be learned is embedded. Another aspect of the thesis of procedural embedding is that the process of going from general goal oriented language which is capable of accomplishing some task to a special purpose, efficient, algorithm for the task should itself be mechanized. By expressing the properties of the special purpose algorithm in terms of their procedural analogues, we can use the analogues to establish that the special purpose routine does in fact do what it is intended to do.

We are concerned as to how a theorem prover can unify structural, problem solving methods with domain dependent algorithms and data into a coherent problem solving process. By structural methods we mean those that are concerned with the formal structure of the argument rather than with the semantics of its domain dependent content.

An example of a structural method is the "consequences of the consequent" heuristic. By the CONSEQUENCES OF THE CONSEQUENT heuristic, we mean that a problem solver should look at the consequences of the goal that is being attempted in order to get an idea of some of the statements that could be useful in establishing or rejecting the goal.

We need to discover more powerful structural methods. PLANNER is intended to provide a computational basis for expressing structural methods. One of the most important ideas in PLANNER is that it brings some of the structural methods of problem solving out into the open where they can be analyzed and generalized. There are a few basic patterns of looping and recursion that are in constant use among programmers. Examples are recursion on binary trees as in LISP

and the FIND statement of PLANNER. The primitive FIND will construct a list of all the objects with certain properties. For example we can find five things which are on something which is green by evaluating

```
<FIND 5 x
      <GOAL (ON x y)>
      <GOAL (GREEN y)>>
```

which reads "find 5 x's such that x is ON y and y is GREEN."

The patterns of looping and recursion represent common structural methods used in programs. They specify how commands can be repeated iteratively and recursively. One of the main problems in getting computers to write programs is how to use these structural patterns with the particular domain dependent commands that are available. It is difficult to decide which, if any, of the basic patterns is appropriate in any given problem. The problem of synthesizing programs out of canned loops is formally identical to the problem of finding proofs using mathematical induction. We have approached the problem of constructing procedures out of goal oriented language from two directions. The first is to use canned loops (such as the FIND statement) where we assume a priori the kind of control structure that is needed. The second approach is to try to abstract the procedure from protocols of its action in particular cases.

Another structural method is progressive refinement. The way problems are solved by progressive refinement is by repeated evaluation. Instead of trying to do a complete search of the problem space all at once, repeated refinements are made. For example in a game like chess the same part of the game tree might be looked at several times. Each time certain paths are more deeply explored in the light of what other searches have revealed to be the key features of the position. Problems in design seem to be particularly suitable for the use of progressive refinement since proposed designs are often themselves amenable to successive refinement. The way in which progressive refinement typically is done in PLANNER is by repeated evaluation. Thus the expression which is evaluated to solve the problem will itself produce as its value an expression to be evaluated.

2. Information Processing Overview

Some readers will prefer to read section 3 which has concrete examples before the abstract discussion in this section.

There are many ways in which one can approach a description of PLANNER. In this section we will describe PLANNER from an Information Processing Viewpoint. To do this we will describe the data structure and the control structure of the formalism.

DATA STRUCTURE:

ASSOCIATIVE MEMORY forms the basis for PLANNER's data space which consists of directed graphs with labeled arcs. The operation of PUTTING and GETTING the components of data objects have been generalized to apply to any data type whatsoever. For example to PUT the value CANONICAL on the expression $(+ X Y (* X Z))$ under the indicator SIMPLIFIED is one way to record that $(+ X Y (* X Z))$ has been canonically simplified. Then the degree to which an expression is simplified can be determined by GETTING the value under the indicator SIMPLIFIED of the expression. The operation of PUT and GET can be implemented efficiently using hash coding. Lists and vectors have been introduced to gain more efficiency for common special case structures. The associative memory is useful to PLANNER in many ways. Monitoring gives PLANNER the capability of trapping all read, write, and execute references to a particular data object. The monitor (which is found under the indicator MONITOR) of the data object can then take any action that it sees fit in order to handle the situation. The associative memory can be used to retrieve the value of an identifier I of a process p by GETTING the I component of p . Code can be commented by simply PUTTING the actual comment under the indicator COMMENT.

DATA BASE: What is most distinctive about the way in which PLANNER uses data is that it has a data base in which data can be inserted and removed. For example inserting (AT B1 P2) into the data base might signify that block B1 is at the place P2. A coordinate of an expression is defined to be an atom in some position. An expression is determined by its coordinates. Assertions are stored in buckets by their coordinates using the associative memory in order to provide efficient retrieval. In addition a

total ordering is imposed on the assertions so that the buckets can be sorted. Imperatives as well as declaratives can be stored in the data base. We might assert that whenever an expression of the form (At object1 place1) is removed from the data base, then any expression in the data base of the form (ON object1 object2) should also be removed from the data base. The data base can be tree structured so that it is possible to simultaneously have several local data bases which are incompatible. Furthermore assertions in the data base can have varying scopes so that some will last the duration of a process while others are temporary to a subroutine.

CONTROL STRUCTURE: PLANNER uses a pattern directed multiprocess backtrack control structure to tie the operation of its primitives together.

BACKTRACKING: PLANNER processes have the capability of backtracking to previous states. A process can backtrack into a procedure activation (i.e. a specific instance of an invocation of the procedure) which has already returned with a result. Using the theory of comparative schematology, we have proved in the dissertation that the use of backtrack control enables us to achieve effects that a language (such as LISP) which is limited to recursive control structure cannot achieve. Backtracking cuts across the subroutine structure of PLANNER. Backtrack control allows the consequences of elaborate tentative hypotheses to be explored without losing the capability of rejecting the hypotheses and all of their consequences. A choice can be made on the basis of the available knowledge and if it doesn't work, a better choice can be made using the new information discovered while investigating the first choice. Also backtrack control makes PLANNER procedures easier to debug since they can be run backwards as well as forwards enabling a problem solver to "zero in" on bugs.

MULTIPROCESSING gives PLANNER the capability of having more than one locus of control in problem solving. By using multiple processes, arbitrary patterns of search through a conceptual problem space can be carried out. Processes can have the power to create, read, write, interrupt, resume, single step, and fork other processes.

PATTERN DIRECTION combines aspects of control and data structure. The fundamental principle of pattern directed computation is that a procedure should be a pattern of what the procedure is intended to accomplish. In other words a procedure should not only do the right thing but it should appear to do the right thing as well! PLANNER uses pattern direction for the following operations:

CONSTRUCTION of structured data objects is accomplished by templates. We can construct a list whose first element is the value of x and whose second element is the value of y by the procedure $(x\ y)$. If x has the value 3 and y has the value (A B) then $(x\ y)$ will evaluate to (3 (A B)).

DECOMPOSITION is accomplished by matching the data object against a structured pattern. If the pattern $(x_1\ x_2)$ is matched against the data object ((3 4) A) then x_1 will be given the value (3 4) and x_2 will be given the value A.

RETRIEVAL: An assertion is retrieved from the data base by specifying a pattern which the assertion must match and thus bind the identifiers in the pattern. For example we can determine if there is anything in the data base of the form (ON $x\ A$). If (ON B A) is the only item in the data base, then x is bound to B. If there is more than one item in the data base which matches a retrieval pattern, then an arbitrary choice is made. The fact that a choice was made is remembered so that if a simple failure backtracks to the decision, another choice can be made.

INVOCATION: Procedures can be invoked by patterns of what they are supposed to accomplish. For example a procedure might be defined which attempts to satisfy patterns of the form (ON $x\ y$) by causing x to be ON y . Such a procedure could be invoked by making (ON A B) a goal. The procedure might or might not succeed in achieving its goal depending on the environment in which it was called. Since many theorems might match a goal, a recommendation is allowed as to which of the candidate theorems might be useful. The recommendation is a pattern which a candidate theorem must match in order to be invoked.

$(+ X Y\ (* X Z))$

SIMPLIFIED

CANONICAL

EXAMPLE OF
AN ASSOCIATION

3. An Extended example

This section contains an extended description of a simple example in PLANNER. It is partially based on a draft written by T. Winograd for the course 6.545. If the reader would like to see a more logically systematic presentation, he can consult the the author's dissertation.

The easiest way to understand PLANNER is to watch how it works, so in this section we will present a few simple examples and explain the use of some of its most elementary features. These examples are not intended to represent TOY PROBLEMS to serve as test cases for "general problem solvers". The toy problem paradigm is misleading because toy problems can be solved without any real knowledge of the domain in which the toy problem is posed. Indeed, it seems gauche to use any thing as powerful as real knowledge on such simple problems. In contrast we believe that real world problems require vast amounts of procedural knowledge for their solution. We see it as part of our task to provide the intellectual capabilities needed for effective problem solving. We would like to see the toy problem paradigm replaced with an INTELLECTUAL CAPABILITY paradigm where the object is to illustrate the intellectual capabilities needed so that knowledge can be effectively embedded in procedures.

First we will take the most venerable of traditional deductions:

```
Turing is a human
All humans are fallible
so
Turing is fallible.
```

It is easy enough to see how this could be expressed in the usual logical notation and handled by a uniform proof procedure. Instead, let us express it in one possible way to PLANNER by saying:

```
<ASSERT (HUMAN TURING)>
<ASSERT <DEFINE THEOREM1
  <CONSEQUENT (Y) (FALLIBLE ?Y)
  <GOAL (HUMAN ?Y)>>>>
```

Function calls are enclosed between "<" and ">". The proof would be generated by asking PLANNER to evaluate the expression:

```
<GOAL (FALLIBLE TURING)>
```

We immediately see several points. First, there are at least two different kinds of information stored in the data base: declaratives and imperatives. Notice that for complex sentences

containing quantifiers or logical connectives we have a choice whether to express the sentence by declaratives or by imperatives.

Second, one of the most important points about PLANNER is that it is an evaluator for statements. It accepts input in the form of expressions written in the PLANNER language and evaluates them, producing a value and side effects. ASSERT is a function which, when evaluated, stores its argument in the data base of assertions. In this example we have defined a theorem of the CONSEQUENT type (we will see other types later). This states that if we ever want to establish a goal of the form (FALLIBLE ?Y), we can do this by accomplishing the goal (HUMAN ?Y), where Y is a identifier. The strange prefix character "?" is part of PLANNER's pattern matching capabilities (which are extensive and make use of the pattern-matching language MATCHLESS which is explained in chapter 4 of the dissertation). If we ask PLANNER to prove a goal of the form (A Y), there is no obvious way of knowing whether A and Y are constants (like TURING and HUMAN in the example) or identifiers. LISP solves this problem by using the function QUOTE to indicate constants. In pattern matching this is inconvenient and makes most patterns much bulkier and more difficult to read. Instead, PLANNER uses the opposite convention -- a constant is represented by the atom itself, while a identifier must be indicated by adding an appropriate prefix. This prefix differs according to the exact use of the identifier in the pattern, but for the time being let us just accept "?" as a prefix indicating a identifier. The definition of the theorem indicates that it has one identifier, Y by the (Y) following CONSEQUENT.

The third statement illustrates the function GOAL, which tries to prove an assertion. This can function in several ways. If we had asked PLANNER to evaluate <GOAL (HUMAN TURING)> it would have found the requested assertion immediately in the data base and succeeded (returning as its value some indicator that it had succeeded). However, (FALLIBLE TURING) has not been asserted, so we must resort to theorems to prove it. Later we will see that a GOAL statement can give PLANNER various kinds of advice on which theorems are applicable to the goal and should be tried. For the moment, take the default case, in which the evaluator tries all theorems whose consequent is of a form which matches the goal (i.e.

a theorem with a consequent (?? TURING) would be tried, but one of the form (HAPPY ??) or (FALLIBLE ?Y ??) would not). Assertions can have an arbitrary list structure for their format -- they are not limited to two-member lists or three-member lists as in these examples.) The theorem we have just defined would be found, and in trying it, the match of the consequence to the goal would cause the identifier Y to be bound to the constant TURING. Therefore, the theorem sets up a new goal (HUMAN TURING) and this succeeds immediately since it is in the data base. In general, the success of a theorem will depend on evaluating a PLANNER program of arbitrary complexity. In this case it contains only a single GOAL statement, so its success causes the entire theorem to succeed, and the goal (FALLIBLE TURING) is proved. The following is the protocol of the evaluation:

```

      <GOAL (FALLIBLE TURING)>
(FALLIBLE TURING) is not in the
data base so attempt to invoke a
theorem to establish the goal
      enter THEOREM1
      Y becomes TURING
      <GOAL (HUMAN TURING)>
is satisfied since the goal is in the
data base
      return (FALLIBLE TURING)

```

The way in which identifiers are bound by matching is of key importance to PLANNER. Consider the question "is anything fallible?", or in logic (EXISTS X (FALLIBLE X)). This could be expressed in PLANNER as:

```

<THPROG (X) <GOAL (FALLIBLE ?X)>>
Notice that THPROG (PLANNER's
equivalent of a LISP PROG, complete
with GO statements, tags, RETURN, etc.)
in this case it acts as an existential
quantifier. It provides a binding-
place for the identifier X, but does
not initialize it -- it leaves it in a
state particularly marked as
unassigned. To answer the question,
we ask PLANNER to evaluate the entire
THPROG expression above. To do this it
starts by evaluating the GOAL
expression. This searches the data
base for an assertion of the form
(FALLIBLE ?X) and fails. It then looks
for a theorem with a consequent of that
form, and finds the theorem we defined
above. Now when the theorem is called,
the identifier Y in the theorem is
linked to the identifier X in the goal,
but since X has no value yet, Y does
not receive a value. The theorem then
sets up the goal (HUMAN ?Y) with Y as
an identifier. The PLANNER primitive

```

GOAL uses the data-base searching mechanism to look for any assertion which matches that pattern (i.e. an instantiation), and finds the assertion (HUMAN TURING). This causes Y (and therefore X) to be bound to the constant TURING, and the theorem succeeds, completing the proof and returning the value (FALLIBLE TURING).

There seems to be something missing. So far, the data base has contained only the relevant objects, and therefore PLANNER has found the right assertions immediately. Consider the problem we would get if we added new information by evaluating the statements:

```

<ASSERT (HUMAN SOCRATES)>
<ASSERT (GREEK SOCRATES)>

```

Our data base now contains the assertions:

```

(HUMAN TURING)
(HUMAN SOCRATES)
(GREEK SOCRATES)

```

and theorem1:

```

<CONSEQUENT (Y) (FALLIBLE ?Y)
  <GOAL (HUMAN ?Y)>>

```

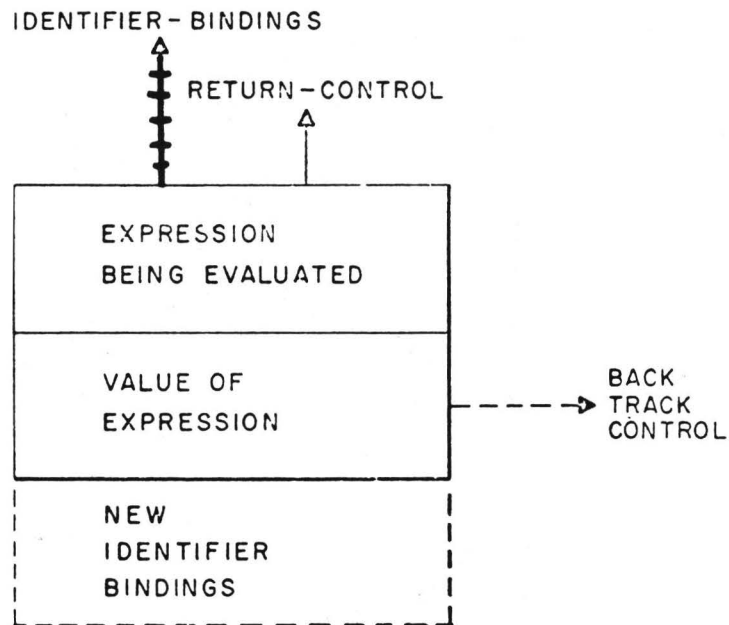
What if we now ask, "Is there a fallible Greek?" In PLANNER we would do this by evaluating the expression:

```

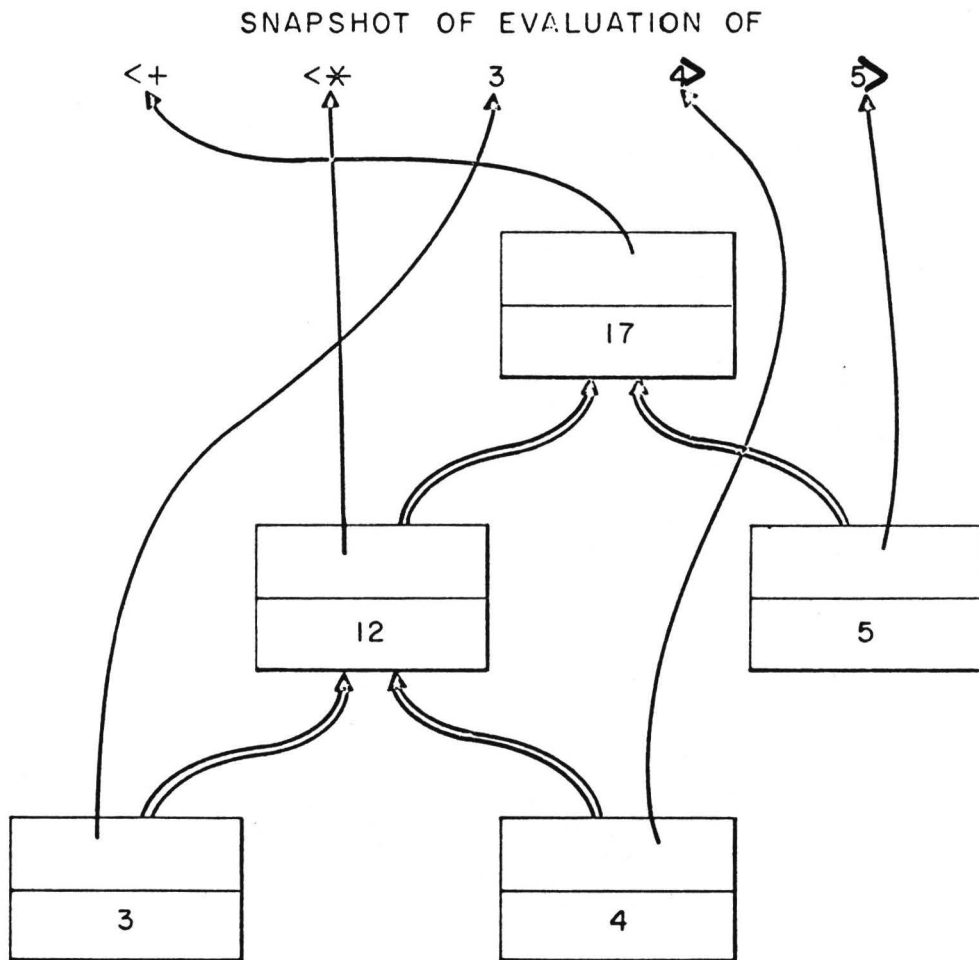
<THPROG (X)
  <GOAL (FALLIBLE ?X)>
  <GOAL (GREEK ?X)>>

```

THPROG is a wishy-washy version of the LISP function PROG. If THPROG runs into a failure trying to evaluate one of the expressions in its body, then it backtracks to the last decision that was made and dumps the responsibility of how to proceed on the procedure which made the decision. Notice what might happen. The first GOAL may be satisfied by exactly the same deduction as before, since we have not removed information. If the data-base searcher happens to run into TURING before it finds SOCRATES, the goal (HUMAN ?Y) will succeed, binding Y and thus X to TURING. After (FALLIBLE ?X) succeeds, the THPROG will then establish the new goal (GREEK TURING), which is doomed to fail since it has not been asserted, and there are no applicable theorems. If we think in LISP terms, this is a serious problem, since the evaluation of the first GOAL has been completed before the second one is called, and the "stack" now contains only the return address for THPROG and the identifier X. If we try

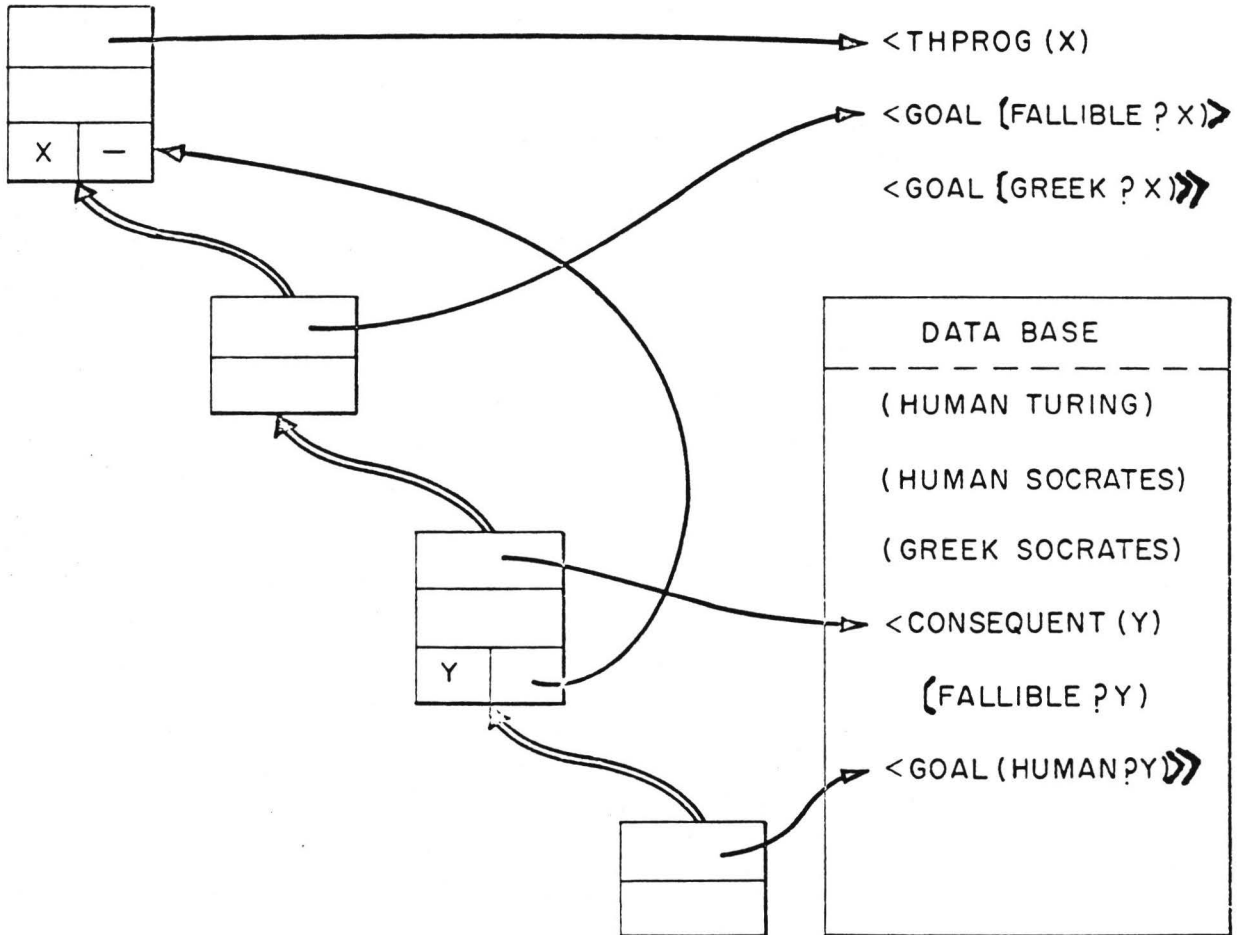
FORMAT OF FUNCTION ACTIVATIONS
IN SNAPSHOTS

NOTE : THE IDENTIFIER-BINDINGS AND RETURN-CONTROL POINTERS OF AN ACTIVATION ARE USUALLY THE SAME AND THUS ARE COMBINED INTO A DOUBLE POINTER LIKE THIS $\Rightarrow$

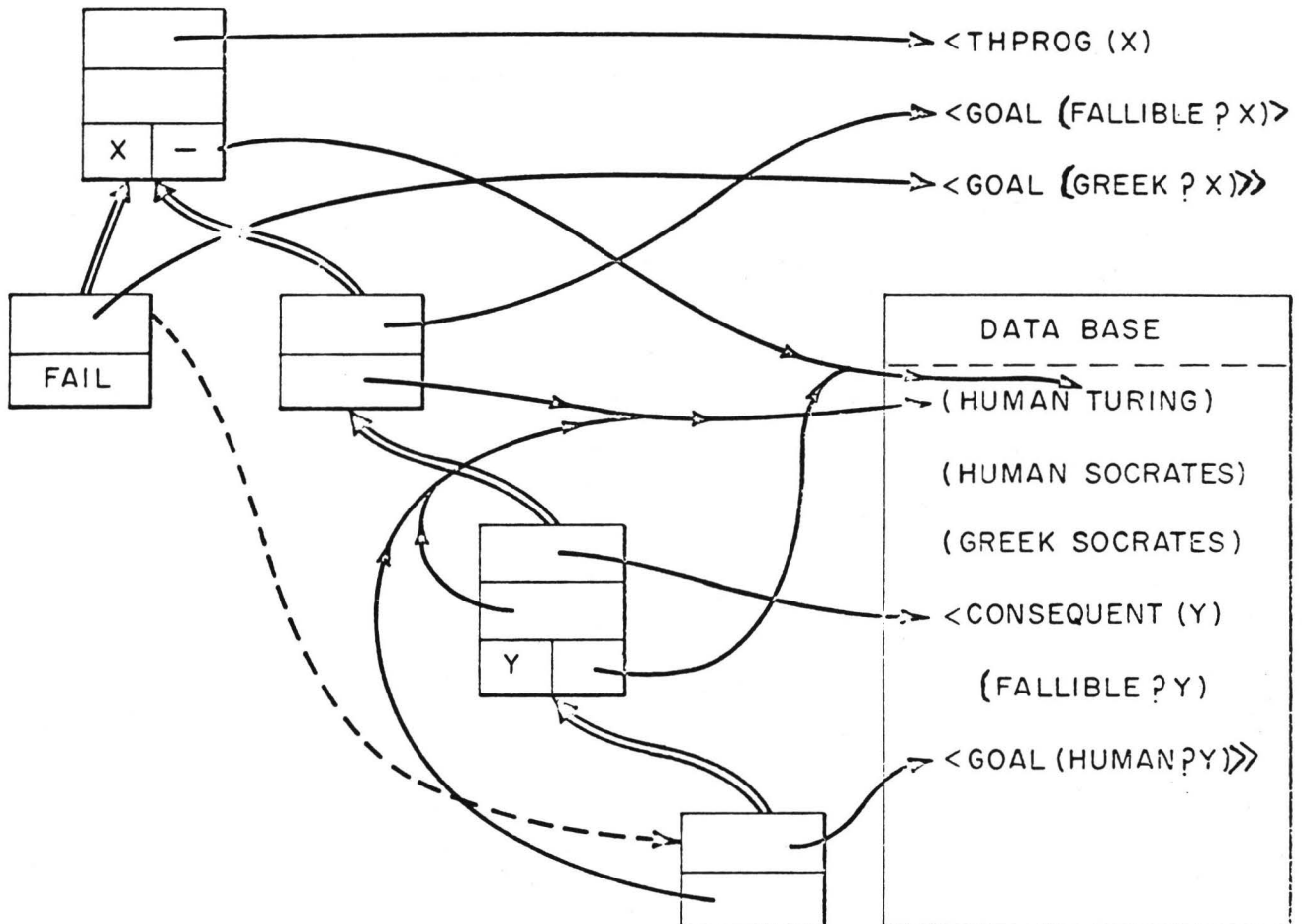


THUS $\langle + \langle * \rangle 3 4 \rangle 5 \rangle$ EVALUATES TO 17.

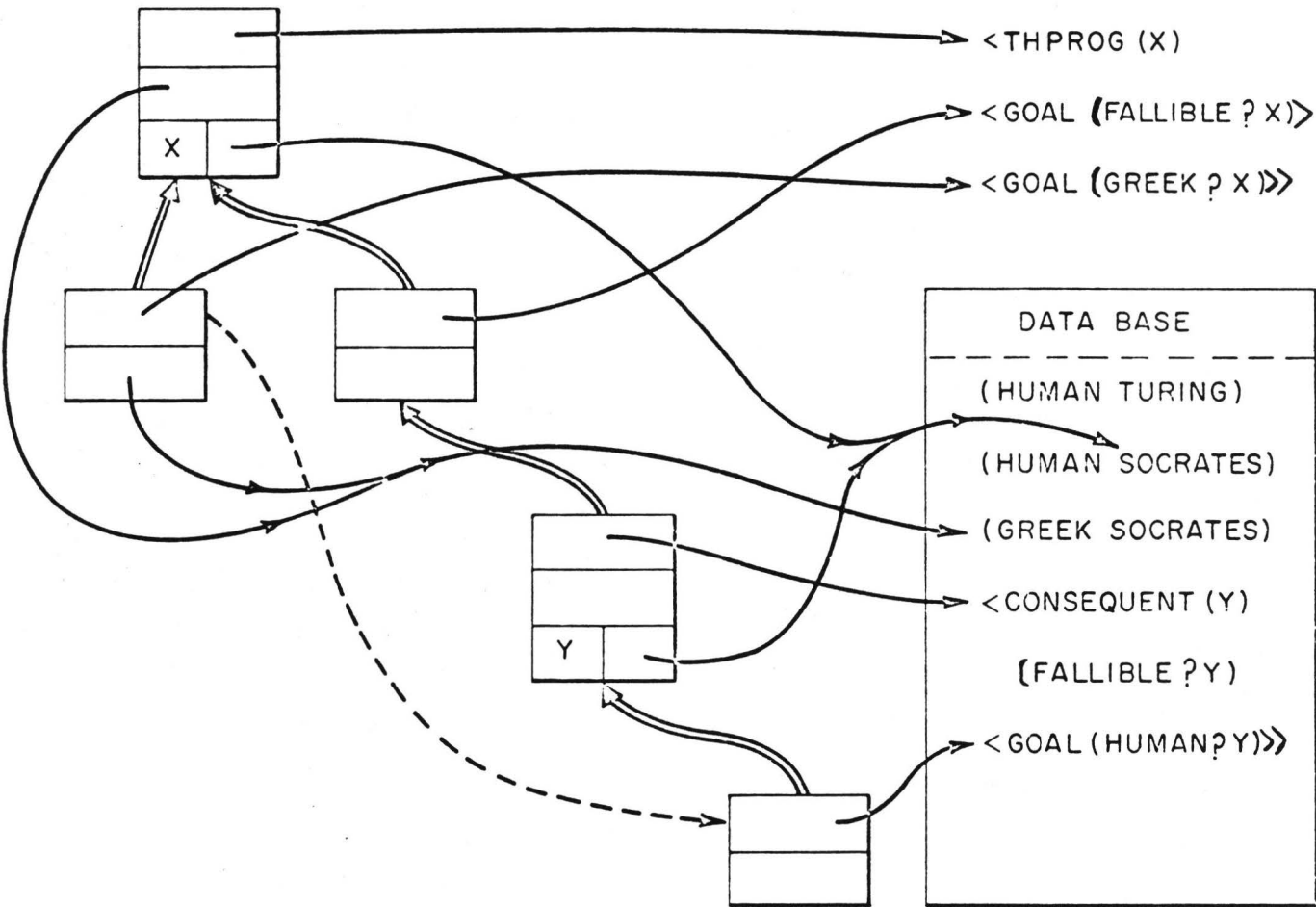
SNAPSHOT NO. 1



SNAPSHOT NO. 2



SNAPSHOT NO. 3



to go back to the beginning and start over, it will again find TURING and so on, ad infinitum.

One of the most important features of the PLANNER language is that backtracking in case of failure is always possible, and moreover this backtracking can go to the last place where a decision of any sort was made. Here, the decision was to pick a particular assertion from the data base to match a goal. Another kind of decision is the choice of a theorem to try to achieve a goal. PLANNER keeps enough information to change any decision and send evaluation back down a new path.

In our example the decision was made inside the theorem for FALLIBLE, when the goal (HUMAN ?Y) was matched to the assertion (HUMAN TURING). PLANNER will retrace its steps, try to find a different assertion which matches the goal, find (HUMAN SOCRATES), and continue with the proof. The theorem will succeed with the value (FALLIBLE SOCRATES), and the THPROG will proceed to the next expression, <GOAL (GREEK ?X)>. Since X has been bound to SOCRATES, this will set up the goal (GREEK SOCRATES) which will succeed immediately by finding the corresponding assertion in the data base. Since there are no more expressions in the THPROG, it will succeed, returning as its value the value of the last expression, (GREEK SOCRATES). The whole course of the deduction process depends on the failure mechanism for backtracking and trying things over (this is actually the process of trying different branches down the conceptual goal tree.) All of the functions like THCOND, THAND, THOR, etc. are controlled by success vs. failure, rather than NIL vs. non-NIL as in LISP. This then is the PLANNER executive which establishes and manipulates subgoals in looking for a proof.

We would now like to give a somewhat more formal description of the behavior of PLANNER on the above problem. If we introduce suitable notation our problem solving protocols can be made much more succinct and their structure made visible. Also by formalizing the notions, we can make PLANNER construct and analyze protocols. This provides one kind of tool by which PLANNER can understand its own behavior and make generalizations on how to proceed.

In this case the protocol is:

```
1: enter THPROG
2:
```

```

X is rebound but not initialized
3:
  <GOAL (FALLIBLE ?X)> will attempt a
  pattern directed invocation since
  nothing in the data base matches
  (FALLIBLE ?X).
4:
  enter THEOREM1
5:
  match (FALLIBLE ?Y) with
  (FALLIBLE ?X) thus linking Y to X
  the situation is shown in
  snapshot number 1
6:
  <GOAL (HUMAN ?Y)> finds
  (HUMAN TURING) in the data base
7:
  Y gets the value TURING
  thus giving X the value TURING
8:
  return (HUMAN TURING)
9:
  THEOREM1 returns (FALLIBLE
  TURING)
10:
  <GOAL (GREEK TURING)> fails since
  it is not in the data base and there
  are no matching consequents

```

Thus PLANNER must backtrack to step 7 and try again. The situation is shown in snapshot number 2. For the convenience of the reader, we will repeat the first six steps from above and then continue the protocol.

```

1: enter THPROG
2:
  X is rebound but not initialized
3:
  <GOAL (FALLIBLE ?X)>
4:
  enter THEOREM1
5:
  match (FALLIBLE ?Y) with
  (FALLIBLE ?X) thus linking Y to X
6:
  <GOAL (HUMAN ?Y)> finds
  (HUMAN TURING) in the data base
11:
  Y gets the value
  SOCRATES thus giving X the value
  SOCRATES
12:
  return (FALLIBLE SOCRATES)
13:
  THEOREM1 returns (FALLIBLE
  SOCRATES)
14:
  <GOAL (GREEK SOCRATES)>
15: return (GREEK SOCRATES) as the top
  level value

```

The situation is shown in snapshot number 3.

So far we have seen that although PLANNER is written as an evaluator, it

differs in several critical ways from anything which is normally considered a programming language. First, it is goal-directed. Theorems can be thought of as subroutines, but they can be called by specifying the goal which is to be satisfied. This is like having the ability to say "Call a subroutine which will achieve the desired result at this point." Second, the evaluator has the mechanism of success and failure to handle the exploration of the conceptual goal tree. In PLANNER there is no explicit goal tree. The conceptual goal tree is represented by a SNAPSHOT of a CONFIGURATION of PROCESSES. Thus PLANNER has powerful control structure primitives to allow the conceptual goal structure to be easily and naturally reflected in the execution of PLANNER processes. Other evaluators, such as LISP, with a basic recursive evaluator have no way to do this. Third, PLANNER contains a large set of primitive commands for matching patterns and manipulating a data base, and for handling that data base efficiently.

On the other side, we can ask how it differs from other theorem provers. What is gained by writing theorems in the form of programs, and giving them power to call other programs which manipulate data? The key is in the form of the data the theorem-prover can accept. Most systems take declarative information, as in predicate calculus. This is in the form of expressions which represent "facts" about the world. These are manipulated by the theorem-prover according to some fixed uniform process set by the system. PLANNER can make use of imperative information, telling it how to go about proving a subgoal, or to make use of an assertion. This produces what is called HIERARCHICAL control structure. That is, any theorem can indicate what the theorem prover is supposed to do as it continues the proof. It has the full power to evaluate expressions which can depend on both the data base and the subgoal tree, and to use its results to control the further proof by making assertions, deciding what theorems are to be used, and specifying a sequence of steps to be followed.

4. Current Applications for PLANNER

The PLANNER formalism is currently being used in a variety of applications. It is being used as part of the conceptual machinery of a robot at M.I.T. and Stanford. The formalism is used for the following purposes in a robot:

- semantic basis for natural language
- formulating and executing plans of action

- finding high level descriptions of visual scenes

- Other applications are its use as a procedural model for:

- architecture design
- children stories
- models of programs
- elementary Euclidean geometry

5. BIBLIOGRAPHY

Balzer, R. EXDAMS - Extendable Debugging and Monitoring System. Proc. SJCC 1969, 34. May, 1969.

Balzer, R. M., On the Future of Computer Program Specification and Organization. December 1970.

Dennis, Jack B. Programming Generality, Parallelism and Computer Architecture. Computation Structures Group Memo No. 32. August 1968.

Earley Jay, Toward an Understanding of Data Structures. Computer Science Department, University of California, Berkeley.

Fikes, R., Ref-Arf: A System for Solving Problems Stated as Procedures Artificial Intelligence (1970).

Fisher, D. A., Control Structures for Programming Languages. 1970.

Hewitt, C., PLANNER: a Language for Proving Theorems, Artificial Intelligence Memo 137, Massachusetts Institute of Technology (project MAC), July 1967.

Hewitt, C. PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot, Proceedings of the International Joint Conference on Artificial Intelligence. Washington D. C. May 1969.

Hewitt, C. Teaching Procedures in Humans and Robots. Conference on Structural Learning. April 5, 1970. Philadelphia, Pa.

Hewitt, C. Description and Theoretical Analysis (using Schemas) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. Phd. Feb. 1971.

Hewitt, C. and Patterson M. Comparative Schematology. Record of Project MAC Conference on Concurrent Systems and Parallel Computation. June 2-5, 1970. Available from ACM.

Kay, Alan C., Reactive Engine, Ph. D. thesis University of Utah, 1970.

McCarthy, J.; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; and Levin, Michael I. 1962. LISP 1.5 Programmer's Manual, M. I. T. Press.

McCarthy, J. and Hayes, P.,

Some Philosophical Problems form the Standpoint of Artificial Intelligence. Stanford A. I. Memo 73.

Newell, A., Shaw, J. C., and Simon, H. A., 1959. Report on a General Problem-solving Program, Proceedings of the International Conference on Information Processing, Paris: UNESCO House, pp. 256-264.

Perlis, A. J. The Synthesis of Algorithmic Systems. JACM. Jan. 1967.

Winograd, T. Procedures as a Representation for Data In a Computer Program for Understanding Natural Language. MAC TR-84. February 1971.