MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 250

December 1971

PLANNER IMPLEMENTATION PROPOSAL TO ARPA 1972-1973

Carl Hewitt

This proposal was written at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70- A-0362-0002.

PLANNER Proposal

Task Objectives

The task objective is the generalization and implementation of the full power of the problem solving formalism PLANNER in the next two years. We will show how problem solving knowledge can be effectively incorporated into the formalism. Several domains will be explored to demonstrate how PLANNER enhances problem solving.

Current Status

Sublanguages of the PLANNER formalism have been implemented at M.I.T. We would now like to have the full power of the formalism available to us. The restrictions of the sublanguages we have now are cramping many of the current applications and discouraging other new applications. The work is currently being pursued on two time-shared computers: a Honeywell 645 and a Digital Equipment Corporation PDP-10. Pursuing work on two systems instead of one is somewhat unusual an so we will give a detailed justification. PLANNER should be independent of any particular computer series or time-sharing monitor. By implementing on two systems we insure that this is true in practice as well as theory. MULTICS and ITS are currently the best systems available for PLANNER. Both systems are currently being upgraded so they will probably still be the best systems in a few years. M.I.T. has signed the contract for the follow on processor for MULTICS. The Stanford A. I. Laboratory is constructing a proto-type for the follow on processor for the PDP-10. Both follow on systems propose to have large amounts of fast random access memory and even larger virtual address spaces. If either one comes to fruition, then PLANNER can be applied to problems with large data bases. PLANNER uses hash coding so that the time that it takes to retrieve a fact is essentially independent of the size of the data base PROVIDING there is enough fast random access memory. One danger to be avoided in this approach is duplicating work on the two systems. So far the interaction between the two implementations has been extremely fruitful.

The following is a detailed comparison of MULTICS and ITS for the purpose of implementing PLANNER:

Fast Random Access Memory: MULTICS has 400K and ITS has 250K. For acceptable response it should be the case that the working sets of all the users fit into fast random access at once. In the day shift both systems often respond slowly. They need either fewer users or more memory.

Address Space: MULTICS has a 36 bit address space which is adequate for our medium term needs. The mapping is implemented by dividing the address space into segments in a way which is sometimes annoying but usually not too harmful. ITS has an 18 bit address space which is entirely inadequate for our needs. The follow on processor for the PDP-10 will have a larger address space. PLANNER uses the address space for dynamic linking, garbage collection, and breathing space between data spaces (especially stacks).

File System: ITS has an adequate file system. MULTICS has a good file system in which the files can be put in the address space of a process which makes it very easy to manipulate files.

Speed: The processors for the two systems execute instructions at approximately the same speed. Because of an administrative decision to keep ITS lightly loaded, it gives much faster response.

Consoles: ITS has consoles that run at a maximum rate of 2400 Baud which is somewhat slow. Multics is currently experimenting with connecting an IMLAC with a 200,000 Baud line. Fast consoles are essential for the use of PLANNER. PLANNER is a unified system in which editing, debugging, monitoring, metering, and executing all take place. It is not necessary or desireable to use a separate editor, compiler, or debugger. Slow consoles would be a real bottleneck in the system.

Next Year

In the next year we would like to complete the design and implementation of extended PLANNER. The design will support the complete formalism as described in Hewitt's thesis. Specifically extensions will be incorporated to encompass multiple states of the world and multiple processes. Multiple states will enable PLANNER to easily and directly compare the situations that would result if alternative plans of action were followed. Multiple processes will enable PLANNER to have more than one locus of problem solving activity in existence at one time. To prove out our ideas we will develop several simple domains such as:

A simple logistics and transportation system with supply depots.

A robot like blocks world with more than one arm so that it is necessary for two processes to cooperate to accomplish some of the tasks.

A negotiator for the game of Diplomacy which would have the knowledge needed to try to survive in a cooperative-competitive world.

A Simple Example

We would like to give an example of how simple requests can be handled in PLANNER. Suppose manufacturer X has just gone bankrupt. Which of our products are affected? In order to get PLANNER to do anything, a goal oriented procedure must be written. Because of the powerful procedure definition machinery in PLANNER arbitrary inferential searches can be carried out. PLANNER is not limited to being able to answer the fixed number of kinds of questions that the system builder happened to think of. The formalism is designed to make it easy for PLANNER procedures to construct other PLANNER procedures. Terry Winograd has shown in his thesis how it is possible to translate English requests like "Find all products which are affected" into the PLANNER procedure:

```
<find all .product
        <goal |affected-by :product X|>>
```

The above PLANNER statement says the problem is to find all products which are affected by X. At this point the system will comment: "I don't know how to find out whether a product is affected by a manufacturer". The user might reply: "A part is affected by a manufacturer if the manufacturer makes it or if it has some subpart which is affected by the manufacturer". Thus the following procedure would be generated by the natural language translator:

```
    <consequent ||
        |affected-by ?part ?manufacturer|
            <or
                <goal |manufactures ?manufacturer ?part|>
                <and
                    <goal |part-of ?subpart ?part|>
                    <goal
                        |affected-by
                            ?subpart
                            ?manufacturer|>>>>
```

The above statements are in a form which can be directly executed by PLANNER. By these means PLANNER can solve problems that were unanticipated by the original system builders. PLANNER will execute the above request efficiently even on a large data base because it indexes its data base using hash coding.

Biographical Note

Carl Hewitt was born in Clinton, Iowa, but considers himself a native of El Paso, Texas to which he moved at the age of two years. He attended El Paso Public Schools and graduated from El Paso High School in 1963. With a McDermott Scholarship, he attended M.I.T. In 1967 he graduated in mathematics, receiving a fellowship to do graduate work in artificial intelligence and theories of computation. His Social Security number is 453-70-1755.

His publications include:

Automata on a Two Dimensional Tape (with Manuel Blum). Annual Conference on Switching and Automata Theory. October 1967. Austin, Texas.

Comparative Schematology (with Michael Paterson). Proceedings of Project MAC Conference on Parallism. June 1970. Woods Hole, Mass.

PLANNER: A Language for Proving Theorems in Robots. Proceedings of IJCAI. May 7-9, 1969. Washington D. C.

Teaching Procedures in Humans and Robots. Proceedings of Conference on Structural Learning. April 5,

1970. Philadelphia, Pa. Journal of Structural Learning.

Procedural Embedding of Knowledge in PLANNER. Proceeding of Second International Joint Conference on Artificial Intelligence. Sept. 1-4, 1971. London.

Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. Phd thesis at M.I.T. Jan. 1971.

Appendix I.

The Structural Foundations of Problem Solving

The following was extracted from chapter 2 of Hewitt's thesis. It gives an overview of PLANNER. Detailed information is available in the Ph. D. theses of Hewitt and Winograd.

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 proveable since a foundation for 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 for 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. As part of the definition of the formalisms, the following elements must be defined: the data structure, the control structure, and the primitive procedures. 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 an active agent called 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 begun 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 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 formulation of a foundation 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 to 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 to say 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.

PLANNER is a high level, goal-oriented formalism in which one can specify to a large degree what one wants done rather than how to do it. Many of the primitives in PLANNER are concerned with manipulating a data base in a pattern directed fashion. Most of the primitives have been developed as extensions to the formalism when we have found problems that could not otherwise be solved in a natural way. Of course the trick is to incorporate the new primitive as a genuine extension of wide applicability. Others have suggested themselves as adjuncts in order to obtain useful closure properties in the formalism. We would be grateful to any reader who could suggest problems that would seem to require further extensions or modifications to the formalism.

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:

GRAPH 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 operations of PUT and GET can be implemented efficiently using hash coding. Lists and vectors have been introduced to gain more efficiency for common special purpose structures. The graph 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 graph 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 graph 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 a procedure) which has already returned with a result. Using the theory of comparative schematology, we have proved in chapter 8 of 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 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 (x1 x2) is matched against the data object ((3 4) A) then x1 will be given the value (3 4) and x2 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 thereby 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.

One basic idea behind PLANNER is to exploit the duality that we find between certain imperative and declarative sentences. Consider the statement (implies A B). The statement is a perfectly good declarative. In addition, it can also have certain imperative uses for PLANNER. It can say that we might set up a procedure which will note whether A is ever asserted and if so to consider the wisdom of asserting B in turn. |Note: it is not always wise! Suppose we assert (integer 0) and (implies (integer n) (integer (+ n 1))|. Furthermore it permits us to set up a procedure that will watch to see if it is ever our goal to try to deduce B and if so whether A should be made a subgoal. Exactly the same observations can be made about the contrapositive of the statement (implies A B) which is (implies (not B) (not A)). Statements with universal quantifiers, conjunctions, disjunctions, etc. can also have both declarative and imperative uses. PLANNER theorems are used as imperatives when executed and as declaratives when used as data. The imperative analogues have the advantage that they can more easily express any procedural knowledge that we might have such as "Don't use this theorem twice".

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 <either 4 <atomic>> 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 an important step 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 plans 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 are collections of procedures for simulating the behavior of the system being modeled. MODELS of PROGRAMS are procedures for defining properties of procedures and attempting to verify the properties so defined. 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 (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.

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

From the above observations, we have constructed a formalism that permits both the imperative and declarative aspects of statements to be easily manipulated. PLANNER uses a pattern-directed information retrieval system. The data base is interrogated by specifying a pattern of what is to be retrieved. Instead of having to explicitly name procedures which are to be called, they can be invoked implicitly by a pattern (this important concept is called PATTERN-DIRECTED INVOCATION). When a statement is asserted, recommendations determine what conclusions will be drawn from the assertion. Procedures can make recommendations as to which theorems should be used in trying to draw conclusions from an assertion, and they can recommend the order in which the theorems should be applied. Goals can be created and automatically dismissed when they are satisfied. Objects can be found from schematic or partial descriptions. Provision is made for the fact that statements that were once true in a model may no longer be true at some later time and that consequences must be drawn from the fact that the state of the model has changed. Assertions and goals created within a procedure can be dynamically protected against interference from other procedures. Unlike some other formalisms such as GPS, PLANNER has no explicit goal tree. Instead the computation itself can be thought to be investigating some conceptual problem space. Primitives for a multiprocess backtrack control structure give flexibility to the ways in which the conceptual problem space can be investigated. Procedures written in the formalism are extendable in that they can make use of new knowledge whether it be primarily declarative or imperative in nature. Hypotheses can be established and later discharged. PLANNER has been used to write a block control language in which we specify how blocks can be moved around by a robot. We would like to write a structure building formalism in which we could provide descriptions of structures (such as houses and bridges) and let PLANNER figure out how to build them. The logical deductive system used by PLANNER is subordinate to the hierarchical control structure of the language. PLANNER theorems operate within a context consisting of return addresses, goals, assertions, bindings, and local changes of state that have been made to the global data base. Through the use of this context we can guide the computation and avoid doing basically the same work over and over again. For example, once we determine that we are working within a group (in the mathematical sense) we can restrict our attention to theorems for working on groups since we have direct control over what theorems will be used. PLANNER has a sophisticated deductive system in order to give us greater power over the direction of the computation. Of course procedures written in PLANNER are not intrinsically efficient. A great deal of thought and effort must be put into writing efficient procedures. PLANNER does provide some basic mechanisms and primitives in which to express problem solving procedures. The control structure can still be used when we limit ourselves to using resolution as the sole rule of inference. A uniform proof procedure gives very little control over how or when a theorem is used. The problem is one of the level of the interpreter that is used. A digital computer by itself will only interpret the hardware instructions of the machine. A higher level interpeter such as LISP will interpret assignments and recursive function calls. At a still higher level an interpreter such as MATCHLESS will interpret patterns for constructing and decomposing strucured data.

PLANNER can interpret assertions, find statements, and goals. It goes without saying that code can be compiled for any of the higher level interpeters so that it actually runs under a lower level interpreter. In general higher level interpreters have greater choice in the actions that they can take since instructions are phrased more in terms of goals to be achieved rather than in terms of explicit elementary actions. The problem that we face is to raise the level of the interpreter while at the same time keeping the actions taken by it under control. Due to its extreme hierarchical control and its ability to make use of new imperative as well as declarative knowledge, it is feasible to carry out very long chains of inference in PLANNER without extreme inefficiency.

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 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 investigation 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 investigations 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 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.

The task of artificial intelligence is to program inanimate machines to perform tasks that require intelligence. Over the past decade several different approaches toward A. I. have developed. Although very pure forms of these approaches will seldom be met in practice, we find that it is useful for purposes of discussion to consider these conceptual extremes. One approach (called results mode by S. Papert) has been to choose some specific intellectual task that humans can perform with facility and write a program to perform it. Several very fine programs have been written following this approach. One of the first was the Logic Theorist which attempted to prove theorems in the propositional calculus using the deductive system developed in Principia Mathematica. The importance of the Logic Theorist is that it developed a body of techniques which when cleaned up and generalized have proved to be fundamental to furthering our understanding of A. I. The results mode approach offers the potentiality of maximum efficiency in solving particular classes of problems. On the other hand, there have been a number of programs written from the results mode approach which have not advanced our understanding although the programs achieved slightly better results than had been achieved before. These programs have been large, clumsy, brute force pieces of machinery. There is a clear danger that the results mode approach can degenerate into trying to achieve A. I. via the "hairy kludge a month plan". The problems with "hairy kludges" are well known. It is impossible to get such programs to communicate with each other in a natural and intimate way. They are difficult to understand, extend, and modify because of the ad hoc way in which they are constructed.

Another approach to A. I. that has been prominent in the last decade is that of the uniform proof procedure. Proponents of the approach write programs which accept declarative descriptions of combinatorial problems and then attempt to solve them. In its most pure form the approach does not permit the machine to be given any information as to how it might solve its problems. The character table approach to A. I. is a modification of the uniform procedure approach in which the program is also given a finite state table of connections between goals and methods. The uniform procedure approach offers a great deal of elegance and a maximum of a certain kind of generality. Current programs that implement the uniform procedure approach suffer from extreme inefficiency. We believe that the inefficiency is intrinsic in the approach.

PLANNER is not necessarily general in the same sense that a uniform proof procedure is general. PLANNER is intended to be a natural computational basis for methods of solving problems in a domain. A complete proof procedure for a quantificational calculus is general in the sense that if one can force the problem into the form of the input language and is prepared to wait eons if necessary then the computer is guaranteed to find a solution if there is one. The approach taken in PLANNER is to subordinate the deductive system to an elaborate hierarchical control structure. Although PLANNER itself is domain independent, procedures written in it have differing overlapping degrees of domain independence. Proponents of the uniform procedure approach are apt to say that PLANNER "cheats" because through the use of its hierarchical control structure, it is possible to tell the program how to try to solve its problems. In order to prevent this kind of "cheating", they would restrict the input to consist entirely of declaratives. But surely, it is to the credit of a program that it is able to accept new imperative information and make use of it. A problem solver needs a high level language for expressing problem solving methods even if the language is only used by the problem solver to express its problem solving methods to itself. PLANNER serves both as the language in which problems are posed to the problem solver and the language in which methods of solution are formulated. PLANNER is not intended to be a solution to the problem of finding general methods for reducing the combinatorial search involved to test whether a given proposition is valid or not. It is intended to be a general formalism in which knowledge in a domain can be combined and integrated. Realistic problem solving programs will need vast amounts of knowledge. We consider all methods of solving problems to be legitimate. If a program should happen to already know the answer to the problem that it is asked to solve, then it is perfectly reasonable for the problem to be solved by table look-up. We should use the criterion that the problem solving power of a program should increase much faster than in direct proportion to the number of things that it is told. The important factors in judging a program are its power, elegance, generality, and efficiency.