MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo No. 203

July 1970

MICRO-PLANNER REFERENCE MANUAL

by

Gerald Jay Sussman and Terry Winograd

This is a manual for the use of the Micro-Planner interpreter, which implements a subset of Carl Hewitt's language, PLANNER and is now available for use by the Artificial Intelligence group

Work reported herein was supported by the Warren McCulloch Laboratory, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0002. Reproduction of this document, in whole or in part, is permitted for any purpose of the United States Government.

Micro-Planner Reference Manual

by
Gerald Jay Sussman and Terry Winograd

I. Introduction

1) Micro-Planner is an implementation of a subset of Carl Hewitt's language, PLANNER, by Gerald Jay Sussman, Terry Winograd, and Eugene Charniak on the AI group computer in LISP. Micro-Planner is now a publically accessible systems program in the AI group system ITS. The current version of Micro-Planner, embedded in an allocated LISP, may be obtained by incanting ':PLNR' or 'PLNR<control-H>' to DDT. Micro-Planner is also available as EXPR code or LAP code. All questions, suggestions, or comments about Micro-Planner should be directed to Gerald Jay Sussman (login name GJS) who will maintain the program.

2) Philosophical Overview

The 'level' of a programming language is a term which is in fairly common usage in the computer culture. We all agree, for example, that FORTRAN, LISP, and PL1 are higher level languages than, say assembly language, regardless of our opinions of their relative merits for expressing our individual styles of programming. After some introspection, I decided that what is meant by the level of a language is the amount of knowledge of programming style implicit in the language processor and thus the quantity of detail which the user does not explicitly have to specify because he may assume that the processor will fill in for him. When we express the value judgement that LISP or PL1, for example, is a lousy language, what we mean is that the stylistic assumptions made by the designers of the language are incompatible with our own styles and thus overly restrict us. There are some of us (may ITS preserve their file directories.) with whom any restrictions are incompatible and who thus always write in assembly language. There are others, such as I, who find such restrictions an aid in the organization of large programs. Thus, to me, LISP is useful in that it provides a large library of subroutines with uniform calling sequences, a convenient evaluator for defining recursively reentrant procedures from those primitives and a bookkeeper for the maintenance of a list structured data base. Indeed for these conveniences I pay heavily in space and time efficiency, as well as in some feeling of constrained style.

PLANNER is the first of perhaps a whole new level of languages which will basically be oriented toward the accomplishment of tasks (or goals) which may in fact be broken down into subtasks (subgoals). By contrast, in previous languages (as LISP) problem solutions are expressed in terms of procedures (functions). The distinction is not immediately apparent but should become more clear if we note that in a PLANNER program, if a goal is activated then it may be satisfied by any number of objects in the data base or by any number of theorems (the analogue of procedure). A backup mechanism is provided so that one of the possibilities is tried, and then if, as a consequence a failure occurs, then another is tried, etc. Furthermore, the data and theorems need not be referenced explicitly, but rather by saying, in essence, 'the datum (or procedure) which fits the following pattern (or accomplishes the desired result)'. In order for such data and theorems to be implicitly referenced efficiently the system has the responsibility of constructing and accessing a cross-referenced data base of data and theorems.

II. Programming in Micro-Planner

The easiest way to understand Micro-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.

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 Micro-Planner by saying:

```
   (THASSERT (HUMAN TURING))
   (DEFPROP THEOREM1
     (THCONSE (X) (FALLIBLE $?X)
                  (THGOAL (HUMAN $?X)))
    THEOREM)
   (THASSERT THEOREM1)
```

The proof would be generated by asking Micro-Planner to evaluate the expression:

```
     (THGOAL (FALLIBLE TURING)  (THTBF THTRUE))
```

We immediately see several points. First, there are two different ways of storing information. Simple assertions are stored in a data base of assertions, while more complex sentences containing quantifiers or logical connectives are expressed in the form of theorems.

Second, one of the most important points about Micro-Planner is that it is an evaluator for statements written in a programming language. It accepts input in the form of expressions written in the Micro-Planner language, and evaluates them, producing a value and side effects. THASSERT is a function which, when evaluated, stores its argument in the data base of assertions or the data base of theorems (which are cross-referenced in various ways to give the system efficient look-up capabilities). A theorem is defined with DEFPROP as are functions in LISP. In this example we have defined a theorem of the THCONSE type (THCONSE means consequent; we will see other types later). This states that if we ever want to establish a goal of the form (FALLIBLE $?X), we can do this by accomplishing the goal (HUMAN $?X), where X is a variable. The strange prefix characters are part of Micro-Planner's pattern matching capabilities. If we ask Micro-Planner to prove a goal of the form (A X), there is no obvious way of knowing whether A and X are constants (like TURING and HUMAN in the example) or variables. 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, Micro-Planner uses the opposite convention -- a constant is represented by the atom itself, while a variable must be indicated by adding an appropriate prefix. This prefix differs according to the exact use of the variable in the pattern, but for the time being let us just accept $? as a prefix indicating a variable. The definition of the theorem indicates that it has one variable, X by the (X) following THCONSE.

The fourth statement illustrates the function THGOAL, which calls the Micro-Planner interpreter to try to prove an assertion. This can function in several ways. If we had asked Micro-Planner to evaluate (THGOAL (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 THGOAL statement can give Micro-Planner various kinds of advice on which theorems are applicable to the goal and should be tried. For the moment, (THTBF THTRUE) is advice that causes the evaluator to try all theorems whose consequent is of a form which matches the goal. (i.e. a theorem with a consequent ($?Z TURING) would be tried, but one of the form (HAPPY $?Z) or (FALLIBLE $?Y $?Z) would not. Assertions can not have an arbitrary list structure for their format but 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 variable $?X to be assigned 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 Micro-Planner program of arbitrary complexity. In this case it contains only a single THGOAL statement, so its success causes the entire theorem to succeed, and the goal (FALLIBLE TURING) is proved.

Consider the question 'Is anything fallible?', or in logic (EXISTS (Y)(FALLIBLE Y)). This requires a variable and it could be expressed in Micro-Planner as:

```
(THPROG (Y) (THGOAL (FALLIBLE $?Y)))
```

Notice that THPROG (Micro-Planner's equivalent of a LISP PROG, complete with GO statements, tags, RETURN, etc.) acts as an existential quantifier. It provides a binding-place for the variable Y, but does not initialize it -- it leaves it in a state particularly marked as unassigned. To answer the question, we ask Micro-Planner to evaluate the entire THPROG expression above. To do this it starts by evaluating the THGOAL expression. This searches the data base for an assertion of the form (FALLIBLE $?Y) 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 variable X in the theorem is identified with the variable Y in the goal, but since Y has no value yet, X does not receive a value. The theorem then sets up the goal (HUMAN $?X) with X as a variable. The data-base searching mechanism takes this as a command to look for any assertion which matches that pattern (i.e. an instantiation), and finds the assertion (HUMAN TURING). This causes X (and therefore Y) to be assigned 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 Micro-Planner has found the right assertions immediately. Consider the problem we would get if we added new information by evaluating the statements:

```
   (THASSERT (HUMAN SOCRATES))
   (THASSERT (GREEK SOCRATES))
```

Our data base now contains the assertions:

```
  (HUMAN TURING)
  (HUMAN SOCRATES)
  (GREEK SOCRATES)
   and the theorem:
  (THCONSE (X) (FALLIBLE $?X)
               (THGOAL (HUMAN $?X)))
```

What if we now ask, ´Is there a fallible Greek?´ In Micro-Planner we would do this by evaluating the expression:

```
   (THPROG (X) (THGOAL (FALLIBLE $?X)) (THGOAL (GREEK $?X)))
```

THPROG acts like an AND, insisting that all of its terms are satisfied before the THPROG is happy. Notice what might happen. The first THGOAL may be satisfied by the exact 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 $?X) will succeed, assigning $?X to TURING. After (FALLIBLE $?X) succeds, 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 THGOAL has been completed before the second one is called, and the ´push-down list´ now contains only the THPROG. If we try 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 Micro-Planner language is that backup in case of failure is always possible, and moreover this backup 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. Other decisions might be the choice of a theorem to satisfy a goal, or a decision of other types found in more complex Micro-Planner functions. Micro-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 $?X) was matched to the assertion (HUMAN TURING). Micro-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, (THGOAL

(GREEK $?X)). Since X has been assigned 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 backing up and trying things over (this is actually the process of trying different branches down the subgoal tree.) All of the functions like THCOND, THAND, THOR, etc. are controlled by success vs. failure. Thus it is the Micro-Planner executive which establishes and manipulates subgoals in looking for a proof.

Although Micro-Planner is written as a programming language, 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 abilitiy 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 subgoal tree. Other evaluators, such as LISP, with a basic recursive evaluator have no way to do this. Third, Micro-Planner contains a bookkeeping system for matching patterns and manipulating a data base, and for handling that data base efficiently.

How is Micro-Planner different from a theorem prover? 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 <u>declarative</u> 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. Micro-Planner can make use of <u>imperative</u> information, telling it how to go about proving a subgoal, or to make use of an assertion. This produces what is called <u>hierachical</u> <u>control</u> <u>structure</u>. That is, any theorem can indicate what the theorem prover is supposed to do as it continues the proof. It has the full power of a general programming language to evaluate functions 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. What does this mean in practical terms? In what way does it make a ´better´ theorem prover? We will give several examples of areas where the approach is important.

First, consider the basic problem of deciding what subgoals to try in attempting to satisfy a goal. Very often, knowledge of the subject matter will tell us that certain methods are very likely to succeed, others may be useful if certain other conditions are present, while others may be possibly valuable, but not likely. We would like to have the ability to use heuristic programs to determine these facts and direct the theorem prover accordingly. It should be able to direct the search for goals and solutions in the best way possible, and able to bring as much intelligence as possible to bear on the decision. In Micro-Planner this is done by adding to our THGOAL statement a <u>recommendation list</u> which can specify that ONLY certain theorems are to be tried, or that certain ones are to be tried FIRST in a specified order. Since theorems are programs, subroutines of any type can be called to help make this decision before establishing a new THGOAL. Each theorem has a name (in our definition on page 1, the theorem was given the name THEOREM1), to facilitate referring to them explicitly.

An important problem is that of maintaining a data base with a reasonable amount of material. Consider the first example above. The statement that all humans are fallible, while unambiguous in a declarative sense is actually ambiguous in its imperative sense (i.e. the way it is to be used by the theorem prover). The first way is to simply use it whenever we are faced with the need to prove (FALLIBLE $?X). Another way might be to watch for a statement of the form (HUMAN $?X) to be asserted, and to immediately assert (FALLIBLE $?X) as well. There is no abstract logical difference, but the impact on the data base is tremendous. The more conclusions we draw when information is asserted, the easier proofs will be, since they will not have to make the additional steps to deduce these consequences over and over again. However since we don't have infinite speed and size, it is clearly folly to think of deducing and asserting everything possible (or even everything interesting) about the data when it is entered. If we were working with totally abstract meaningless theorems and axioms (an assumption which would not be incompatible with many theorem-proving schemes), this would be an insoluble dilemma. But Micro-Planner is designed to work in the real world, where our knowledge is much more structured than a set of axioms and rules of inference. We may very well, when we assert (LIKES $?X POETRY) want to deduce and assert (HUMAN $?X), since in deducing things about an object, it will very often be relevant whether that object is human, and we shouldn't need to deduce it each time. On the other hand, it would be silly to assert (HAS-AS-PART $?X SPLEEN), since there is a horde of facts equally important and equally limited in use. Part of the knowledge which Micro-Planner should have of a subject, then, is what facts are important, and when to draw consequences of an assertion. This is done by having theorems of an antecedent type:

```
  (DEFPROP THEOREM2
        (THANTE (X Y) (LIKES $?X $?Y)
                      (THASSERT (HUMAN $?X)))
        THEOREM)
```

This says that when we assert that X likes something, we should also assert (HUMAN $?X). Of course, such theorems do not have to be so simple. A fully general Micro-Planner program can be activated by an THANTE theorem, doing an arbitrary (that is, the programmer has free choice) amount of deduction, asssertion, etc. Knowledge of what we are doing in a particular problem may indicate that it is sometimes a good idea to do this kind of deduction, and other times not. As with the CONSEQUENT theorems, Micro-Planner has the full capacity when something is asserted, to evaluate the current state of the data and proof, and specifically decide which ANTECEDENT theorems should be called.

Micro-Planner therefore allows deductions to use all sorts of knowledge about the subject matter which go far beyond the set of axioms and basic deductive rules. Micro-Planner itself is subject-independent, but its power is that the deduction processs never needs to operate on such a level of ignorance. The programmer can put in as much heuristic knowledge as he wants to about the subject, just as a good teacher would help a class to understand a mathematical theory, rather than just telling them the axioms and then giving theorems to prove.

Another advantage in representing knowledge in an imperative form is the use of a theorem prover in dealing with processes involving a sequence of events. Consider the case of a robot manipulating blocks on a table. It might have data of the form, ´block1 is on block2,´ ´block2 is behind block3´, and ´if x is on y and you put it on z, then x is on z, and is no longer on y unless y is the same as z´. Many examples in papers on theorem provers are of this form (for example the classic ´monkey and bananas´ problem). The problem is that a declarative theorem prover cannot accept a statement like (ON B1 B2) at face value. It clearly is not an axiom of the system, since its validity will change as the process goes on. It must be put in a form (ON B1 B2 S0) where S0 is a symbol for an initial state of the world. The third statement might be expressed as:

```
(FORALL (X Y Z S)(AND (ON X Y (PUT X Y S))
                      (OR(EQUAL Y Z)
                         (NOT(ON X Z (PUT X Y S))))))
```

In this representation, PUT is a function whose value is the state which results from putting X on Y when the previous state was S. We run into a problem when we try to ask (ON Z W (PUT X Y S)) i.e. is block Z on block W after we put X on Y? A human knows that if we haven´t touched Z or W we could just ask (ON Z W S) but in general it may take a complex deduction to decide whether we have actually moved them, and even if we haven´t, it will take a whole chain of deductions (tracing back through the time sequence) to prove they haven´t been moved. In Micro-Planner, where we specify a process directly, this whole type of problem can be handled in an intuitively more satisfactory way by using the primitive function THERASE.

Evaluating (THERASE (ON $?X $?Y)) removes the assertion (ON $?X $?Y) from the data base. If we think of theorem provers as working with a set of axioms, it seems strange to have function whose purpose is to erase axioms. If instead we think of the data base as the ´state of the world´ and the operation of the prover as manipulating that state, it allows us to make great simplifications. Now we can simply assert (ON B1 B2) without any explicit mention of states. We can express the necessary theorem as:

```
   (DEFPROP THEOREM3
     (THCONSE (X Y Z) (PUT $?X $?Y)
      (THGOAL (ON $?X $?Z))
      (THERASE (ON $?X $?Z))
      (THASSERT (ON $?X $?Y)))
    THEOREM)
```

This says that whenever we want to satisfy a goal of the form (PUT $?X $?Y), we should first find out what thing Z the thing X is sitting on, erase the fact that it is sitting on Z, and assert that it is sitting on Y. We could also do a number of other things, such as proving that it is indeed possible to put X on Y, or adding a list of specific instructions to a movement plan for an arm to actually execute the goal. In a more complex case, other interactions might be involved. For example, if we are keeping assertions of the form (ABOVE $?X $?Y) we would need to delete those assertions which became false when we erased (ON $?X $?Z) and add those which became true when we added (ON $?X $?Y). ANTECEDENT theorems would be called by the assertion (ON $?X $?Y) to take care of that part, and a similar group called ERASING theorems can be called in an exactly analogous way when an assertion is erased, to derive consequences of the erasure. Again we emphasize that which of such theorems would be called is dependent on the way the data base is structured, and is determined by knowledge of the subject matter. In this example, we would have to decide whether it was worth adding all of above relations to the data base, with the resultant need to check them whenever something is moved, or instead to omit them and take time to deduce them from the ON relation each time they are needed.

Thus in Micro-Planner, the changing state of the world can be mirrored in the changing state of the data base, avoiding any need to make explicit mention of states, with the requisite overhead of deductions. This is possible since the information is given in an imperative form, specifying theorems as a series of specific steps to be executed.

If we look back to the distinction between assertions and theorems made on the first page, it would seem that we have established that the base of assertions is the 'current state of the world', while the base of theorems is our permanent knowledge of how to deduce things from that state. This is not exactly true, and one of the most exciting possibilities in Micro-Planner is the capability for the program itself to create and modify the Micro-Planner functions which make up the theorem base. Rather than simply making assertions, a particular Micro-Planner function might be written to put together a new theorem or make changes to an existing theorem, in a way dependent on the data and current knowledge. It seems likely that meaningful 'learning' involves this type of behavior rather than simply modifying parameters or adding more individual facts (assertions) to a declarative data base.

III. The Micro-Planner Primitives

This section will basically be a list of the Micro-Planner primitives with a detailed description of each. Meta-Linguistic variables will be enclosed in angle brackets (<>).

The heart of Micro-Planner is a structure known as THTREE; it is to the hierarchical control structure of PLANNER what a push-down list is to a recursive control structure such as is found in LISP. In LISP the push down list remembers the return addresses of recursive function calls; in Micro-Planner THTREE keeps track of the decisions (or hypotheses) made so far in the problem-solving process which are currently considered relevant to the solution. In case a failure occurs PLANNER can back up THTREE, undoing the decisions which caused the failure until a promising approach is found. If none is found the program returns a failure. THTREE is a tree structure, each node of which contains information about how to proceed in case either success or failure propagates to that node. Failure is propagated from a node if and only if a failure propagates to it and no further possibilities exist at that node. A node of THTREE may be thought of as a goal, with branches originating at this node associated with tentatively useful hypotheses for establishing the goal.

Closely associated with THTREE is THALIST, the list of variable bindings. Certain primitives bind variables by declaration. In their initial bound state variables are called unassigned and are assigned to 'THUNASSIGNED.' THALIST shares the tree structure of THTREE.

Because of the non-recursive implementation of the Micro-Planner interpreter the PLANNER value of an expression is not the same as its LISP value. In most cases this is unimportant because PLANNER expressions are usually executed for effect rather than for value. The PLANNER value of an expression is however available as the LISP value of the LISP free variable THVALUE immediately after the expression is executed. Although this is not a very pretty convention it introduces some very great simplifications in the implementation.

1) (THPROG <variable declaration> <e1> ... <en>)

THPROG is the PLANNER equivalent of the LISP function PROG. Its job is to bind the variables mentioned in the declaration and then to execute the expressions <ei> in sequence, unless changes in sequence are specified by tags and THGO statements. As in LISP, atoms occuring in THPROG bodies are interpreted as tags. If (THGO <tag>) is executed at any point in the interpretation of a THPROG, execution then proceeds from the expression immediately following the tag <tag>. THGO statements may refer to tags which are not in the current THPROG but which are in one which called it. The execution of THPROG terminates either with a failure, successful execution of its last expression, <en>, or by a forced success with a THRETURN statement. (THRETURN <exp>) is equivalent to (THSUCCEED THPROG <exp>) and will cause the THPROG to succeed and return as its PLANNER value the LISP value of the indicated expression. If a THPROG returns by succeeding past the last statement, or by executing a (THSUCCEED THPROG), it returns as its PLANNER value the atom THNOVAL. If it fails, it returns NIL.

The variable declaration list is a list of variable declarations. A variable declaration is either an atom which is the name of a variable to be used inside the THPROG, or a list of two elements, the first of which is the variable name and the second of which is a LISP expression whose LISP value is to be used as the initial value of the variable. If a variable is bound without giving it an initial value it is given the default value 'THUNASSIGNED.'

Evaluation of a THPROG begins at the first expression of the indicated sequence. If it succeeds, a new branch is generated on THTREE and the next expression in sequence is evaluated, etc. If any statement fails then the branches are unwound until either a new success ends the failure propagation or there are no more branches and the THPROG fails. Thus THPROG behaves as a THAND with variable binding capability; it fails unless each of its subexpressions succeeds, allowing for backup, until it returns. As usual, any LISP expression may appear in a THPROG; if it evaluates to NIL, a failure is generated, otherwise a success is assumed. Other relevant functions are THSUCCEED, THFAIL, THFINALIZE, THGO, and THRETURN.

2) (THAND <e1> ... <en>)

THAND fails unless each of its subexpressions succeeds in sequence, allowing for backup. It is just like THPROG except that there are no variable declarations or tags allowed.

3) (THOR <e1> ... <en>)

THOR succeeds if at least one of its subexpressions succeeds. Basically, it CDRs down the list looking for a winner and if it finds one it succeeds, returning its value as the PLANNER value. If a failure propagates back to it, however, it continues CDRing from the point it left off until it finds another winner or it loses.

4) (THAMONG <variable name> <expression>)

Upon entry the variable named (by an atom) must be bound. If it is unassigned then it will be assigned to the first member of the list of choices to which <expression> THVALuates. If the variable already has a value (is assigned), THAMONG fails unless the assigned value is already among the choices. Each time a failure backs up to the THAMONG the variable will be assigned to the next element of the choice list. If it runs out of choices it fails, otherwise it succeeds.

5) (THGOAL <pattern> <rec1> ... <recn>)

This is a real dilly to describe. It is probably the most complex single primitive in Micro-Planner. Assume the simplest case in which there are no recommendations <reci> given. THGOAL searches the data base for a datum (i.e. an assertion) which matches the pattern. If it finds one, it succeeds after assigning all of the unassigned variables in the pattern so as to make it match the datum; it then returns the assertion found as its PLANNER value. If it does not find a matching datum it fails. If after a success, a failure propagates back to it, it unassigns the variables it assigned last time and continues its search for a matching datum from where it left off.

If recommendations are given they are tried in order. If the very first recommendation is a (THNODB) or a (THDBF -) the initial data base search is inhibited, otherwise it is assumed in default. The possible recommendations are:

1) (THNODB) - Inhibit data base search. If it is not first it is useless and causes an error.

2) (THDBF <filter>) - Try only those elements of the data base satisfying the filter.

3) (THTBF <filter>) - Try only those theorems satisfying the filter.

4) (THUSE <th1> <th2> ...<thn>) - Try the theorems given explicitly by their atom headers in the order mentioned.

A filter is any unary LISP predicate; the always true predicate is supplied by the system as THTRUE. All assertions have property lists which are their CDRs. Thus if a filter refers to the CDR of its argument it is referring to the property list. CAR of an assertion, however, is not -1 as in LISP atoms; rather it is the datum which is matched against the pattern.

If a theorem is recommended, say with THUSE, it had better be an atom with a THEOREM property pointing to a consequent theorem (i.e. it must be of the form (THCONSE <variable declarations> <consequent> <e1> ... <en>))). If the goal pattern matches the consequent, then the theorem will be tried. The matcher will first bind the theorem variables appearing in the declaration (see THPROG) and then match the patterns, causing some of the theorem variables to be assigned and leaving others unassigned. If the match wins, the theorem will proceed to execute as a THPROG. If the theorem succeeds the PLANNER value of the THGOAL will be the pattern of the goal with the assignments substituted for the assigned variables, unless the theorem does a THRETURN, in which case the PLANNER value of the THGOAL will be the goodie returned. If a goal variable which is unassigned is matched against a theorem variable and the theorem eventually gives that variable a value then the goal variable also gets the value. A more detailed description of the matcher will be given later.

6) (THFIND <mode> <skeleton> <variable declarations> <e1> ... <en>)

THFIND is a primitive whose THVALuation yields a list of objects, each of which is the result of substituting for variables in the skeleton values of those variables which cause the program starting at the variable declarations (like a THPROG) to succeed. Thus, for example, the data-base contained the assertions (HACKER N) (HACKER H) (HACKER RG) (AT MAC RG) and we THVALed the expression (THFIND ALL (AT SC $?X) (X) (THGOAL (HACKER $?X)) (THNOT (THGOAL (AT MAC $?X)))) we would get ((AT SC N) (AT SC H)) as its THVALUE (PLANNER value). The mode field of a THFIND statement is a triplet (<at least> <at most> <result>). THFIND fails unless it finds at least a number equal to CAR of the mode and if it finds greater than or equal to CADR of the mode it returns CADDR of the mode. If CADR of the mode is not a number it will never be found. The mode triplet may be abbreviated by ALL = (1 NIL NIL), any number n = (n n T).

7) (THCOND <pair1> ... <pair n>)
THCOND is the PLANNER analogue of COND in LISP. As in MAC-LISP the 'pairs' needn't be. Basically THCOND executes the CAR of each pair until one succeeds. The THCOND will then succeed if all the rest of that 'pair' succeeds (like a THAND) else THCOND will fail.

8) (THNOT <e>) is defined as (THCOND (<e> (THFAIL)) ((THSUCCEED))).

9) (THASSERT <skeleton> <rec1> ... <recn>)
THASSERT adds the assertion (formed by substituting assignments for variables (or by THVALing $E's as described in the matcher section) in the skeleton) to the data-base except if the skeleton is an atom, in which case it adds the atom as a theorem to the theorem base.
THASSERT only fails if it tries to assert an already existing assertion. If the first recommendation to a THASSERT is a (THPROP <e>) then the LISP value of <e> is used as the property list of the assertion being asserted. THASSERT also may recommend antecedent theorems with THTBF or THUSE as in THGOAL, though the success or failure of those theorems is irrelevant to success or failure of the assertion. The PLANNER value of a THASSERT is the object asserted.

10) (THERASE <skeleton> <rec1> ... <recn>) is identical to THASSERT except for effect. If a failure backs up to an assertion or an erasure it is undone.

11) (THDO <e1> ... <en>)
THDO executes each of its subexpressions in turn and cares not whether they succeed or fail; it then succeeds. If a failure backs up to it, all that it did is undone.

12) (THAPPLY <theorem> <datum>) calls the specified theorem causing it to match its pattern to the specified datum. If it matches the theorem is executed with <datum> as its 'argument.' The THVALUE of a THAPPLY is the value of the theorem applied.

13) THPUTPROP, THREMPROP, THRPLACA, THRPLACD are just like their LISP counterparts except that if a failure backs up to them they undo their effect.

14) (THSETQ <var1> <e1> ... <varn> <en>)
Sets variable 1 to the value of e1 and ... and sets variable n to the value of en. Undone if failure backs up to it. If the variable is a planner variable then the corresponding expression is THVALed; otherwise the expression is EVALed. See warning about THVAL.

15) (THVSETQ <var1> <e1> ... <varn> <en>)
Sets the variables to the values of eis as in THSETQ. Not undone on failure backup. See warning about THVAL.

16) (THV <variable name>) (THNV <variable name>)
These LISP functions get the PLANNER value of the variable whose name is given. The atoms THV and THNV also serve as markers of the special variable flags to the matcher. (THV FOO) is $?FOO and (THNV FOO) is $_FOO by using the macro-character feature of LISP 1.6.

17) (THVAL <expression> <alist>)
THVAL is the Micro-Planner evaluator just as EVAL is the LISP evaluator. As in LISP, the first argument is evaluated (THVALuated) with free variables in it given the values associated with them on the given alist. Micro-Planner's alist, called THALIST, is a list of pairs (CAR-CADR not CAR-CDR) of variable names and values. If you want to stop a Micro-Planner evaluation at the next interruptable place hit ctrl-A. Planner will then type out ↑A-THVAL and go into its listen loop. The next expression to be executed is in the variable THE. To proceed, type T<space>; to terminate the calculation by infinite failure type NIL<space>. Warning: an explicit call to THVAL may not be reentered for backup upon failure after that call returns.

18) (THUNIQUE <variable name>)
THUNIQUE is a primitive by wich a user may test to see if he is in one kind of infinite loop. THUNIQUE assumes that the variable given is bound. If the variable is bound, it is assumed that it is assigned to a list of atoms and expressions. THUNIQUE will then fail if there is any previous binding and assignment of the given variable for which the atoms and expressions THVALuate identically, the atoms beng interpreted as variables.

19) (THFINALIZE <arg1> <arg2>)
THFINALIZE is a primitive which allows pruning of THTREE. Essentially, if one THFINALIZES, say to a tag, then all the things done since that tag was passed are not undoable in case of failure. Example: To put all of the green blocks in the box and return a list of those actually moved:

```
(THPROG (X (Y NIL))
   (THOR (THAND (THGOAL (IS $?X BLOCK))
                (THGOAL (COLOR $?X GREEN)))
         (THRETURN $?Y))
   FOO
   (THCOND ((THGOAL (CONTAIN BOX $?X)) (THFAIL))
           ((THGOAL (PUTIN $?X BOX) (THUSE TC-PUTIN))
            (THSETQ $?Y (CONS $?X $?Y))
            (THFINALIZE THTAG FOO)
            (THFAIL))
           ((PRINT $?X) (THERT CAN NOT PUT IT IN))))
```

Besides finalizing to a tag, one can finalize a theorem or a THPROG by saying (THFINALIZE THEOREM) or (THFINALIZE THPROG). There are lots of other things that can be done with THFINALIZE, but I will not guarantee them.

20) (THSUCCEED <arg1> <arg2>)
THSUCCEED caused success to propagate, the extent of which is determined by the arguments:
(THSUCCEED THTAG <tag>) = (THGO <tag>) - see THPROG.
(THSUCCEED THPROG <e>) = (THRETURN <e>) - see THPROG.
(THSUCCEED <place>) = (THSUCCEED <place> T)
(THSUCCEED THEOREM <e>) causes theorem to succeed with value of <e>.
(THSUCCEED THEOREM) causes the theorem to succeed with value THNOVAL.
(THSUCCEED) causes a simple success to propagate.

21) (THFAIL <arg1> <arg2> <arg3>)
THFAIL causes failure to propagate, the extent of which is determined by the arguments:
(THFAIL THTAG <tag> T) causes a failure to propagate to the tag indicated.
(THFAIL THTAG <tag>) causes a failure to propagate past the tag indicated.
(THFAIL THPROG), (THFAIL THEOREM) cause the THPROG or the THEOREM currently in to fail.
(THFAIL THINF) causes failure to top level of THVAL.
(THFAIL THMESSAGE <message>) causes a failure to propagate until it reaches a THMESSAGE statement whose pattern matches the message.

22) (THMESSAGE <variable declarations> <pattern> <e1> . . . <en>) examines failures propagating to it, if one has a message which matches the pattern (after the declarations are made) control then passes to the body, <ei>, which executes as a THPROG.

23) (THGO <tag>) = (THSUCCEED THTAG <tag>)

24) (THRETURN <e>) = (THSUCCEED THPROG <e>)

25) (THASVAL <variable>)
THASVAL is a predicate which assumes that the indicated variable is bound. It succeeds if and only if the variable is assigned a value.

26) (THFLUSH <indicator1> <indicator2> ...)
THFLUSH is a generally useful function for getting a LISP into some desired state. It remprop's all properties with the indicators specified from all atoms on the OBLIST.

27) (THDUMP) dumps the state of a Micro-Planner as to assertions and theorems on the currently selected output devices.

28) (THDATA) causes Micro-Planner to go into a read loop for gobbling assertions and theorems at high speed. Loop ends when NIL is read.

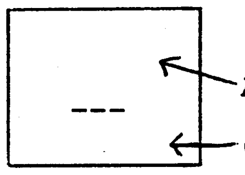
29) (THBKPT <comment>) has no effect unless THTRACEd. If it is, a THBKPT becomes a comment which is typed out and which breaks if desired.

30) THEOREMS
   Theorems are the Micro-Planner analogue of functions in LISP. There are three kinds:
   consequent theorem - for establishing goals
   antecedent theorems - for expanding on assertions
   erasing theorems - for expanding on erasures.
Theorems are defined by (DEFPROP <name> <body> THEOREM) and are added to the data-base using THASSERT or THDATA and may be removed with THERASE. The form of a theorem is (<indicator> <variable declarations> <pattern> <e1> ... <en>), where the indicator may be THCONSE, THANTE, or THERASING, whichever is appropriate. When a theorem is called the declared variables are bound and thee pattern is matched to the calling pattern in THGOAL or datum in THASSERT, THERASE, and THAPPLY. This has the effect of assigning some of the theorem's variables. The theorem is then executed as a THPROG if the match succeeds.

31) The Matcher
   The Matcher in Micro-Planner is elementary and contains a minimum of bells and whistles but is (hopefully) sufficiently powerful for most problems. If more powerful matching capabilities are needed, hooks are provided for the user to hang himself by. (If you wish to hang yourself come see me first. If you really need hair, wait for Hewitt's PLANNER.) Micro-Planner's matcher only matches lists one level deep and only has two distinct kinds of variable occurrences. The matcher's actions can be summarized in the 6-by-6 chart on the next page, which shows how any two pattern elements interact. One nice hook in the matcher is that a pattern element (or the entire pattern) may be of the form $E<expression> or (THEV <expression>) (those macro-characters are at it again.) , which means that the element (or pattern) is to be replaced with the result of THVALuating the expression with an appropriate THALIST. For example, to stack up all red objects:
```
   (THGOAL (STACKUP $E(THFIND ALL $?X (X) (THGOAL (COLOR $?X RED)) ))).
```

MATCHER CHARACTERISTICS

Theorem Pattern \ Calling Pattern		?	$?X or (THV X)		$←X or (THNV X)		Constant C
			un-assigned	assigned	un-assigned	assigned	
?		---	---	---	---	---	---
$?Y or (THV Y)	un-as-signed	---	V(X)←V(Y) Y←X ---	V(Y)←V(X) ---	V(X)←V(Y) Y←X ----	V(X)←V(Y) Y←X ---	V(Y)←C ---
	as-signed	---	V(X)←V(Y) Y←X ---	--- V(Y)=V(X)	V(X)←V(Y) Y←X ---	V(X)←V(Y) Y←X ---	--- V(Y)=C
$←Y or (THNV Y)	un-as-signed	---	V(X)←V(Y) Y←X ---	V(Y)←V(X) ---	V(X)←V(Y) Y←X ---	V(X)←V(Y) Y←X ---	V(Y)←C ---
	as-signed	---	V(X)←V(Y) Y←X ---	V(Y)←V(X) ---	V(X)←V(Y) Y←X ---	V(X)←V(Y) Y←X ---	V(Y)←C ---
Constant D		---	V(X)←D ---	--- V(X)=D	V(X)←D ---	V(X)←D ---	--- D = C



Action Taken -- Blank if no action is taken

Condition for Success -- Blank if always succeeds

V(X) means value of X, V(Y) means value of Y

V(Y)←V(X) means Y assigned to value of X

Y←X means Y assigned to value of X such that if Y is clobbered, so is X

IV. A Compendium of Micro-Planner Error Comments

The Micro-Planner error comments are meant to be self-explanatory. If you come across one which is not, it is probably meant for me, not you, and it would help in removing residual bugs for you to save the situation and show it to me. All Micro-Planner error comments are typed out by a break function called THERT which leaves you in a LISP listen loop at a point as close as possible to the occurrence of the error, so that the state of the system may be interrogated. Usually the error is fatal, but if you wish to proceed, type T<space>. By typing NIL<space> at the listen loop the program will be aborted by infinite failure in an effort to restore the data base. Usually something is typed out before the error comment; that is the object the comment is complaining about. Micro-Planner error comments almost always end with -<funny word>; the <funny word> is the name of the system function which got mad. The authorized list follows:

LISPERROR - THVAL
LISP became unhappy when trying to evaluate the expression typed out.

BAD SUCCEED - THVAL, or BAD FAIL - THVAL
You were screwed by a Planner bug. Please save situation and contact me.

UNCLEAR RECOMMENDATION - THTRY
The expression typed out before the error comment was a recommendation to THGOAL which was not either a THTBF, THDBF, or a THUSE, or was a THNODB which did not come first in the list.

BAD THEOREM - THTRY1
The expression printed was passed to THGOAL as a THEOREM. It either did not have a THEOREM property or was not a consequent theorem.

NOT FOUND - THUNIQUE
The expression printed out was passed to THUNIQUE as the reference variable; unfortunately it was unbound.

THUNBOUND - THUNIQUE
The variable printed out was unbound. It was accessed by THUNIQUE.

BAD CALL - THFINALIZE
You called THFINALIZE without giving a place to finalize to.

OVERPOP - THFINALIZE
THFINALIZE overpopped THTREE trying to find the lousy place to stop.

OVERPOP - THSUCCEED
THSUCCEED overpopped THTREE trying to find the place to stop succeeding. Remember that THGO and THRETURN use THSUCCEED.

OVERPOP - THFAIL
THFAIL overpopped THTREE trying to find a place to stop failing.

IMPURE ASSERTION OR ERASURE - THASS1
The datum you tried to assert or erase had a variable which was unassigned.

BAD THEOREM - THTAE
You tried to call an antecedent or erasing theorem which was of the wrong type or did not have a theorem property.

UNCLEAR RECOMMENDATION - THTAE
You gave a THASSERT or THERASE recommendation which was not a THTBF or a THUSE.

ODD NUMBER OF GOODIES - THSETQ, or ODD NUMBER OF GOODIES - THVSETQ
THSETQ or THVSETQ does not know what to do with the last goodie.

THUNBOUND - THV1
You tried to access the value of the unbound variable printed.

THUNASSIGNED - THV1
You tried to use the value of the unassigned variable printed.

OVERPOP - THREMBIND
Probably a Planner bug. See GJS.

THUNBOUND - THGAL
You tried to access the unbound variable printed.

V. Hints and Kinks

1) All atoms whose pnames begin with TH are property of Micro-Planner and should not be used in programs designed to interact with it.

2) When Micro-Planner is loaded (by :PLNR in DDT) it is listening in a READ-THVAL-PRINT loop at the top level. Typing t<space> will cause this loop to be exited to a LISP READ-EVAL-PRINT loop. Any error condition which propagates to the top level (eg. cntrl-G) will restart the READ-THVAL-PRINT listen loop.

3) The system version of Micro-Planner has a LAP in it. It is desirable to REMLAP at the earliest possible time as the LAP takes up lots of space.

4) The EXPR version of Micro-Planner is in DSK:GJS; PLNR > and the LAP version is stored in DSK:GJS;PLNR LAP.

5) Because most Micro-Planner primitives operate by modifying THTREE and then returning, it is fairly unilluminating to try to trace them with a LISP tracer (which is good for the LISP recursive control structure). For that reason I have taken pains to provide a tracer which is more relevent to the planner control structure. It is available as EXPR code in DSK:GJS;THTRAC >. With it one can conditionally trace and break on theorems, assertions, erasures, and goals. To use the tracer one must first read it in and then incant (THTRACE <obj1> . . . <objn>), where each object is an item to be traced or broken at. The possible entries are either:
<atom>   abbreviation for (<atom> t nil)
(<atom> <trace condition>)   abbreviation for (<atom> <trace condition> nil) or
(<atom> <trace condition> <break condition>)
The CAR of the item may either be the name of a theorem or one of the following atoms which have special meanings:

THEOREM	all theorems
THGOAL	all goals
THASSERT	all assertions
THERASE	all erasures
THBKPT	all breakpoints

The conditions are THVALed with the THALIST which is current at the time of call and thus may be arbitrary PLANNER programs which test the state of variables and the data base. Tracing of selected items, or of all items may be terminated using THUNTRACE.

6) A Micro-Planner program may at any point call a LISP program, but a LISP program may not call a Micro-Planner primitive because the PLANNER control structure is not really recursive. If a LISP routine wishes to call a PLANNER program it must explicitly THVAL it with an appropriate THALIST. Be especially careful not to screw around with the LISP values of Micro-Planner primitives unless you understand what you are doing.

The possibilities for lossage are infinite.