Technical Report 294

Understanding Simple Picture Programs

Ira P. Goldstein

MIT Artificial Intelligence Laboratory

UNDERSTANDING SIMPLE PICTURE PROGRAMS

by

Ira P. Goldstein

Massachusetts Institute of Technology

September 1974

Revised version of a dissertation submitted to the Department of Mathematics on September 4, 1973 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

What are the characteristics of the process by which an intent is transformed into a plan and then a program? How is a program debugged? This paper analyzes these questions in the context of understanding simple turtle programs.

To understand and debug a program, a description of its intent is required. For turtle programs, this is a model of the desired geometric picture. A picture language is provided for this purpose.

Annotation is necessary for documenting the performance of a program in such a way that the system can examine the procedure's behavior as well as consider hypothetical lines of development due to tentative debugging edits. A descriptive framework representing both causality and teleology is developed.

To understand the relation between program and model, the plan must be known. The plan is a description of the methodology for accomplishing the model. Concepts are explicated for translating the global intent of a declarative model into the local imperative code of a program.

Given the plan, model and program, the system can interpret the picture and recognize inconsistencies. The description of the discrepancies between the picture actually produced by the program and the intended scene is the input to a debugging system. Repair of the program is based on a combination of general debugging technique and specific fixing knowledge associated with the geometric model primitives.

In both analyzing the plan and repairing the bugs, the system exhibits an interesting style of analysis. It is capable of debugging itself and reformulating its analysis of a plan or bug in response to self-criticism. In this fashion, it can qualitatively reformulate its theory of the program or error to account for surprises or anomalies.

Thesis Supervisor: Seymour A. Papert
Professor of Applied Mathematics

ACKNOWLEDGEMENTS

This work has grown out of the unique environment for studying human and machine intelligence that has developed at the M. I. T. Artificial Intelligence Laboratory. The author would like to express his gratitude to the many members of the laboratory who encouraged and criticized this research. Special thanks are due to Seymour Papert, Marvin Minsky, Carl Hewitt, Gerry Sussman, David Marr, Patrick Winston, Bruce Roberts, Ben Kuipers, Berthold Horn, Scott Fahlman, Henry Lieberman, Allen Brown, and Gerrianne Goldstein.

The author would also like to express his gratitude to Cheryl Goodman for help in preparing the text and Suzin Jabari for drafting the many illustrations.

ACKNOWLEDGEMENTS

This work has grown out of the unique environment for studying human and machine intelligence that has developed at the M. I. T. Artificial Intelligence Laboratory. The author would like to express his gratitude to the many members of the laboratory who encouraged and criticized this research. Special thanks are due to Seymour Papert, Marvin Minsky, Carl Hewitt, Gerry Sussman, David Marr, Patrick Winston, Bruce Roberts, Ben Kuipers, Berthold Horn, Scott Fahlman, Henry Lieberman, Allen Brown, and Gerrianne Goldstein.

The author would also like to express his gratitude to Cheryl Goodman for help in preparing the text and Suzin Jabari for drafting the many illustrations.

TABLE OF CONTENTS

CHAPTER 1 -- INTRODUCTION

1.1 UNDERSTANDING PROGRAMS

What is the process by which an intent is transformed into a plan and then a program? How is a procedure debugged? This paper analyzes these questions in the context of understanding simple programs for drawing pictures. Figure 1.1 and 1.2 illustrate some typical intended drawings and the the corresponding pictures produced by programs with bugs.

To make concrete our theory of planning and debugging, a computer monitor called MYCROFT has been designed that is able to repair programs written by a beginner. In building such a monitor, fundamental problem solving issues are addressed including simplification, linearity, planning, annotation and self-criticism. This research provides insight into the programming process which is useful both for better educating students and for increasing the competence of machines.

It is important to note at the outset that by "design" I mean that the system has been described in sufficient detail to be hand-simulated but that it has not actually been implemented. The criteria by which the concepts and techniques introduced were judged was their success on a large number of examples of actual programs written by beginners. The system was not implemented to avoid being submerged in details and artificial limitations due to the idiosyncracies of the particular programming language. Also, the primary goal of the research has been to find more precise ways of describing the fundamentals of programming and debugging rather than to construct a practical system for computer-aided instruction. Currently, MYCROFT is being implemented in CONNIVER [McDermott 1972], a LISP-based language with database, pattern-matching and sophisticated control primitives. The implementation is being undertaken in order to provide a platform for further research and will be separately documented in a forthcoming AI memo.

TURTLE PICTURES

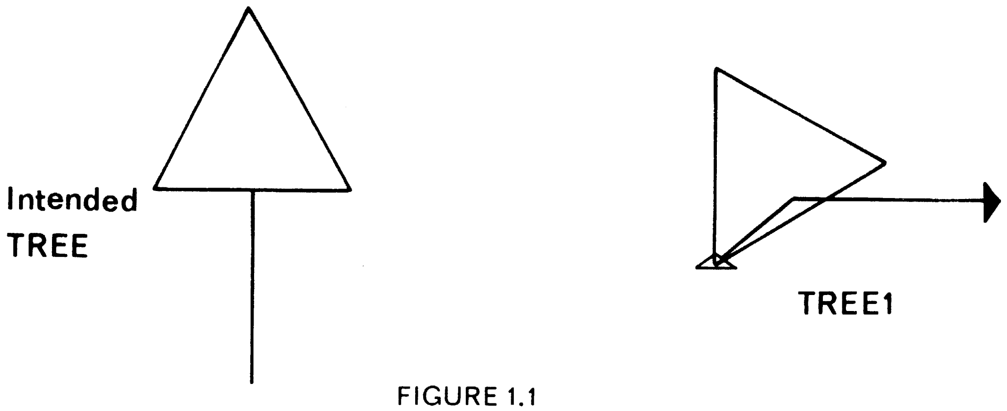


FIGURE 1.1

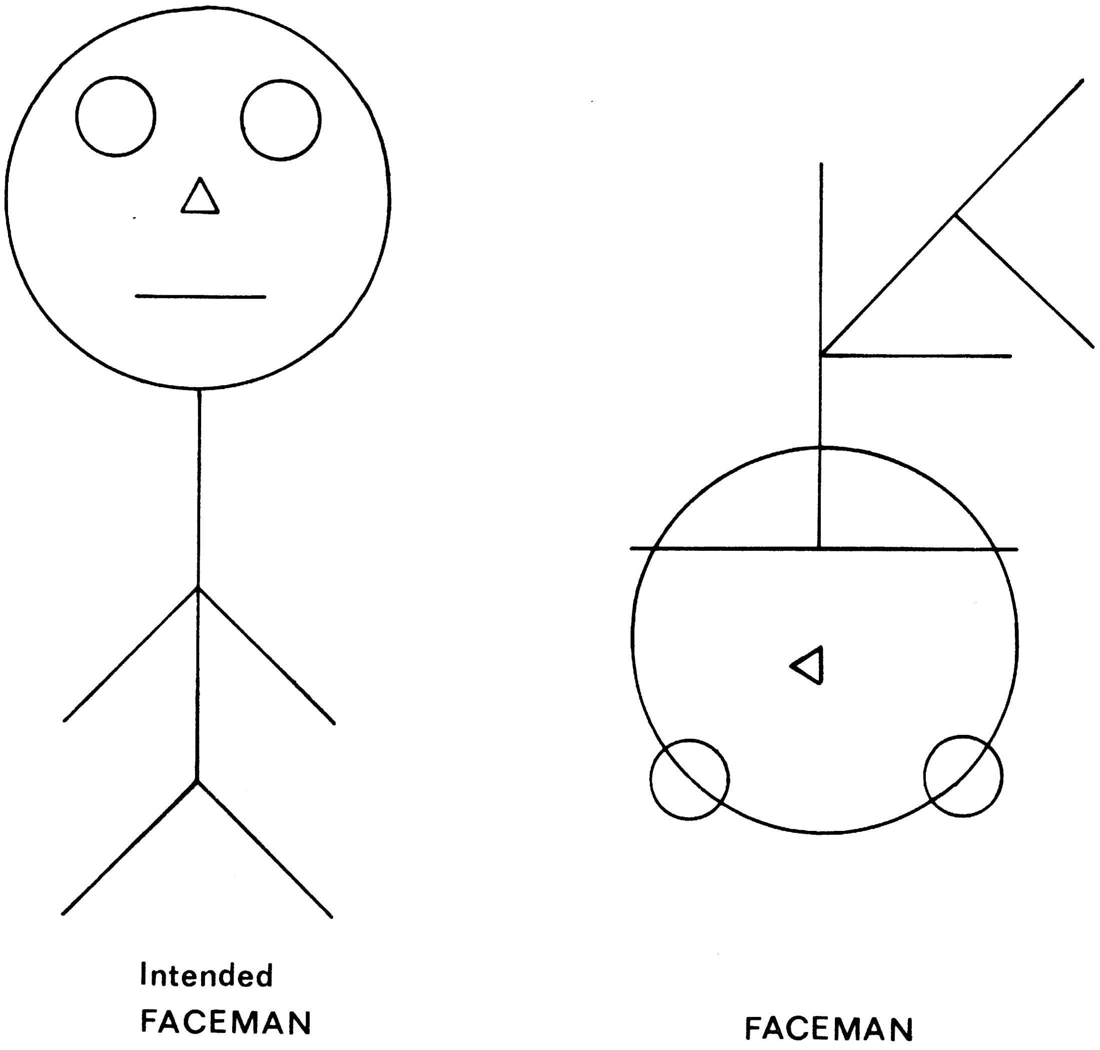


FIGURE 1.2

1.2 TURTLE PROGRAMS AND LOGO

The pictures of figures 1.1 and 1.2 are drawn by program manipulation of a graphic device called the turtle. The turtle has a pen which can leave a track along the turtle's path. Turtles can be either real physical devices or simulations on a graphic display. I shall limit myself to display turtles to avoid problems of imprecision due to motors, drive belts and wheels. Figure 1.3 illustrates the behavior of the turtle executing the following TRIANGLE procedure:

```
TO TRIANGLE
10 FORWARD 100
20 RIGHT 120
30 FORWARD 100
40 RIGHT 120
50 FORWARD 100
END
```

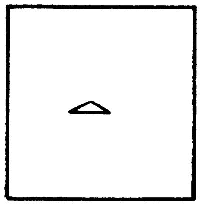
Appendix A provides further details of the turtle language.

Turtles play an important role in the LOGO environment where children learn mathematics and problem solving by programming display turtles, physical turtles with various sensors, and music boxes [Papert 1971a, 1971c, 1972a]. Turtle programs have proven to be an excellent starting point for teaching programming to beginners (of all ages). Hence, in building a system to understand such programs, we can expect to address fundamental issues in the epistemology of procedures.

The turtle programs are expressed in LOGO syntax. LOGO's design goals of clarity and simplicity make it ideal for expressing these programs in a readable way. The syntax, however, is not of significance for the problems of understanding these programs. Their important characteristics are determined by the semantics of turtles.

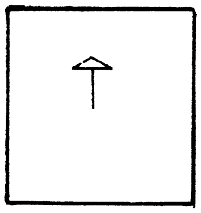
DRAWING A TRIANGLE

1



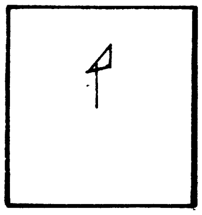
Initial State
TURTLE
AT
HOME

2



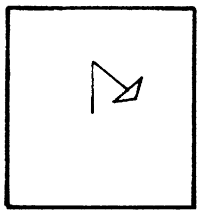
Statement 10
FORWARD 100
Executed

3



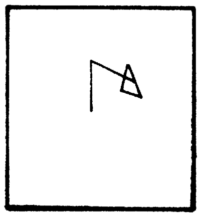
Statement 20
RIGHT 120
Executed

4



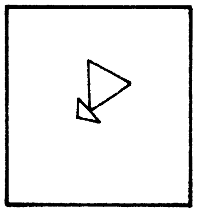
Statement 30
FORWARD 100
Executed

5



Statement 40
RIGHT 120
Executed

6

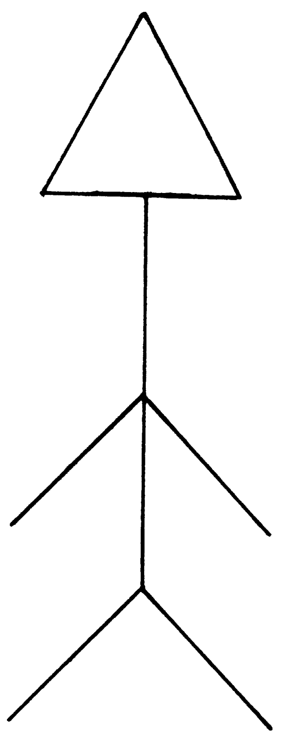


Statement 50
FORWARD 100
Executed

FIGURE 1.3

1.3 PICTURE MODELS

To judge the success of a program, MYCROFT requires as input from the user a description of intent. A declarative language has been designed to define picture models. These models specify important properties of the desired final outcome without indicating the details of the drawing process. The primitives of the model language are geometric predicates for such properties as connectivity, relative position, length and location. The following models are typical of those that the user might provide to describe figure 1.4.



Intended Man

FIGURE 1.4

```
MODEL MAN
M1 PARTS HEAD BODY ARMS LEGS
M2 EQUITRI HEAD
M3 LINE BODY
M4 V ARMS, V LEGS
M5 CONNECTED HEAD BODY, CONNECTED BODY ARMS, CONNECTED BODY LEGS
M6 BELOW LEGS ARMS, BELOW ARMS HEAD
END

MODEL V
M1 PARTS L1 L2
M2 LINE L1, LINE L2
M3 CONNECTED L1 L2 (VIA ENDPOINTS)
END

MODEL EQUITRI
M1 PARTS (SIDE 3) (ROTATION 3)
M2 FOR-EACH SIDE (= (LENGTH SIDE) 100)
M3 FOR-EACH ROTATION (= (DEGREES ROTATION) 120)
M4 CONNECTED (SIDE 1) (SIDE 2)
M5 CONNECTED (SIDE 2) (SIDE 3)
M6 CONNECTED (SIDE 3) (SIDE 1)
END
```

The MAN and V models are underdetermined: they do not describe, for example, the actual size of the pictures nor the specific location of the connection points. The user has latitude in his description of intent because MYCROFT is designed to debug programs that are almost correct. Therefore, not only the model, but also the picture drawn by the program and the definition of the procedure provide clues to the purpose of the program.

1.4 THE NAPOLEON PROGRAMS

To introduce the system, we will examine its performance on a stick figure program and its sub-procedures intended to draw figure 1.4. These programs will be typical of the elementary class of fixed-instruction programs which MYCROFT understands. A fixed-instruction program is one wherein the primitives are restricted to constant inputs. Sub-procedures are allowed; however, no conditionals, variables, recursions, or iterations are permitted.

```
TO NAPOLEON          <- (accomplish man)
10 VEE               <- (accomplish legs)
20 FORWARD 100       <- (accomplish (piece 1 body))
30 VEE               <- (insert arms body)
40 FORWARD 100       <- (accomplish (piece 2 body))
50 LEFT 90           <- (setup heading for head)
60 TRICORN           <- (accomplish head)
END

TO VEE               <- (accomplish v)
10 RIGHT 45          <- (setup heading for l1)
20 BACK 100          <- (accomplish l1)
30 FORWARD 100       <- (retrace l1)
40 LEFT 90           <- (setup heading for l2)
50 BACK 100          <- (accomplish l2)
60 FORWARD 100       <- (retrace l2)
END

TO TRICORN           <- (accomplish equitri)
10 FORWARD 50        <- (accomplish (piece 1 (side 1)))
20 RIGHT 90          <- (accomplish (rotation 1))
30 FORWARD 100       <- (accomplish (side 2))
40 RIGHT 90          <- (accomplish (rotation 2))
50 FORWARD 100       <- (accomplish (side 3))
60 RIGHT 90          <- (accomplish (rotation 3))
70 FORWARD 50        <- (accomplish (piece 2 (side 1)))
END
```

These programs have bugs. VEE draws figure 1.5, TRICORN figure 1.6, and NAPOLEON figure 1.7.

1.5 PLANNING

The "<-" comments shown above constitute the plans for these procedures and explain the purpose of the code in terms of the model. This commentary is essential both for the system to judge the success of the program and to guide the debugging process. From a program-writing point of view, the plan explains how a declarative model of global intent is to be translated into the local imperative code of a program.

The programmer can supply the plan; or, alternatively, MYCROFT can analyze the model and program and deduce the plan. The latter was the case

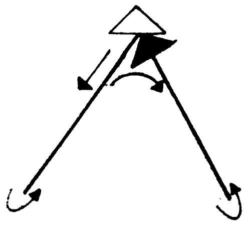

Picture drawn by VEE

FIGURE 1.5

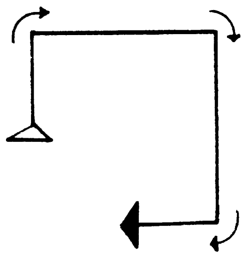

Picture drawn by TRICORN

FIGURE 1.6

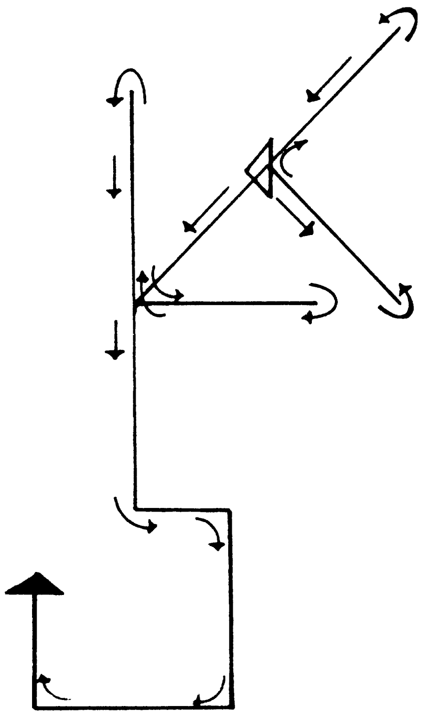

Picture drawn by
NAPOLEON

FIGURE 1.7

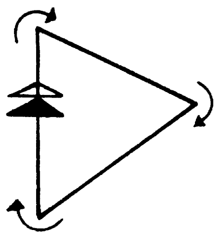

Corrected TRICORN

FIGURE 1.8

for the above programs. The system initially searches for a linear plan. This is a very simple but common type of plan in which the main goals are achieved independently and interactions are limited to interfaces. An interesting characteristic of the plan-finding algorithm is that it is capable of correcting its linear hypothesis in response to anomalies and thereby recognize more complex types of plans.

Planning expertise is obviously essential to both program-writing and debugging. It reveals the user's intent, allowing proper recognition of bugs, and provides guidance on the likely pitfalls of different abstract planning structures. Plans organized around linearity, preparation, interrupts, repetition, and global knowledge will be discussed in chapter 2.

1.6 DEBUGGING

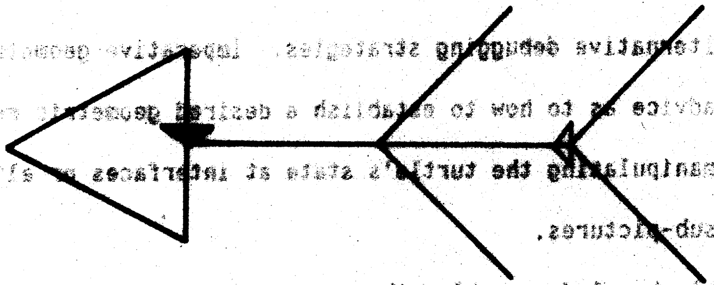
Given the plan, model, and program, the system can interpret the picture and recognize inconsistencies. The description of the discrepancies between the picture actually produced by the program and the intended scene is the input to the debugging system. The system recognizes the following model violations in the NAPOLEON picture.

```
(NOT (LINE BODY))         ;The body is not a line.
(NOT (EQUITRI TRICORN))   ;The head is not an equilateral triangle.
(NOT (BELOW LEGS ARMS))   ;The legs are not below the arms.
(NOT (BELOW ARMS HEAD))   ;The arms are not below the head.
```

Repair of the program is based upon a combination of general debugging techniques and specific imperative knowledge associated with the geometric model primitives. The debugging techniques include the ability to rank violations on the basis of debugging complexity; the ability to analyze errors initially in a modular way but to be prepared to look for "second-order" causes due to interactions; and criteria for choosing between alternative debugging strategies. Imperative geometric knowledge includes advice as to how to establish a desired geometric relation by such means as manipulating the turtle's state at interfaces or altering the scale of sub-pictures.

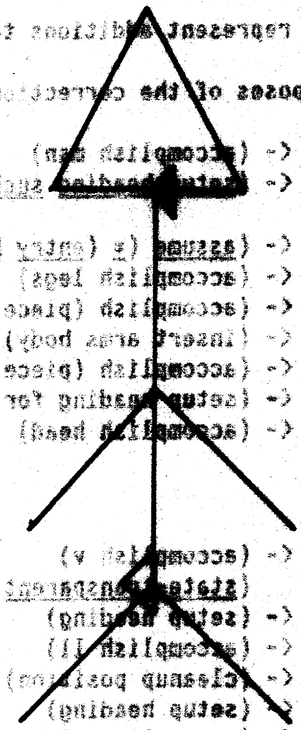
This knowledge enables the system to correct the NAPOLEON, VEE and TRICORN programs. The repair is accomplished in three steps: first the head is corrected (figure 1.8); then the crooked body is straightened (figure 1.9); and finally the orientation of the stick figure is fixed (figure 1.10). The end result is the program shown below which successfully draws the intended picture (figure 1.4). The underlined statements are the corrections made by the debugger. The associated underlined comments represent additions to the plan which explain the assumptions and purposes of the corrections.

```
TO NAPOLEON      <- (accomplish man)
 5 RIGHT 90      <- (setup heading such-that (below legs arms)
                                              (below arms head))
                 <- (assume (= (entry heading) 270))
10 VEE           <- (accomplish legs)
20 FORWARD 100   <- (accomplish (piece 1 body))
30 VEE           <- (insert arms body)
40 FORWARD 100   <- (accomplish (piece 2 body))
50 LEFT 90       <- (setup heading for head)
60 TRICORN       <- (accomplish head)
END


TO VEE           <- (accomplish v)
                    (state-transparent vee)
10 RIGHT 45      <- (setup heading)
20 BACK 100      <- (accomplish l1)
30 FORWARD 100   <- (cleanup position)
40 LEFT 90       <- (setup heading)
50 BACK 100      <- (accomplish l2)
60 FORWARD 100   <- (cleanup position)
70 RIGHT 45      <- (cleanup heading)
END
```



NAPOLEON with Parts Corrected

FIGURE 1.9



Picture Drawn by Corrected NAPOLEON

FIGURE 1.10

```
TO TRICORN      <- (accomplish equitri)
10 FORWARD 50   <- (accomplish (piece 1 (side 1)))
20 RIGHT 120    <- (accomplish (rotation 1))
                <- (= (degrees (rotation 1)) 120)
30 FORWARD 100  <- (accomplish (side 2))
40 RIGHT 120    <- (accomplish (rotation 2))
                <- (= (degrees (rotation 2)) 120)
50 FORWARD 100  <- (accomplish (side 3))
60 RIGHT 120    <- (accomplish (rotation 3))
                <- (= (degrees (rotation 3)) 120)
70 FORWARD 50   <- (accomplish (piece 2 (side 1)))
END
```

1.7 ORGANIZATION

The organization of the monitor system is illustrated in figure 1.11. The main flow of control is represented by the solid lines and is from left to right in the flowchart. The dotted lines indicate advice. The ovals are the major procedures used by MYCROFT and consist of ANNOTATE, FINDPLAN, INTERPRET and DEBUG. The squares contain data. Input to the system consists of the turtle program and picture model. Subsequent processing produces a Cartesian picture description, a plan, a list of violations and finally an edited corrected program.

The next two chapters investigate the nature of planning and debugging in the turtle world. This knowledge comprises the foundation of the research. Subsequent chapters provide details of the algorithms for finding the plan and for annotating the program's performance. The final chapter discusses extensions of the basic system to the analysis of more complex types of programs and to education.

FLOWCHART OF MYCROFT

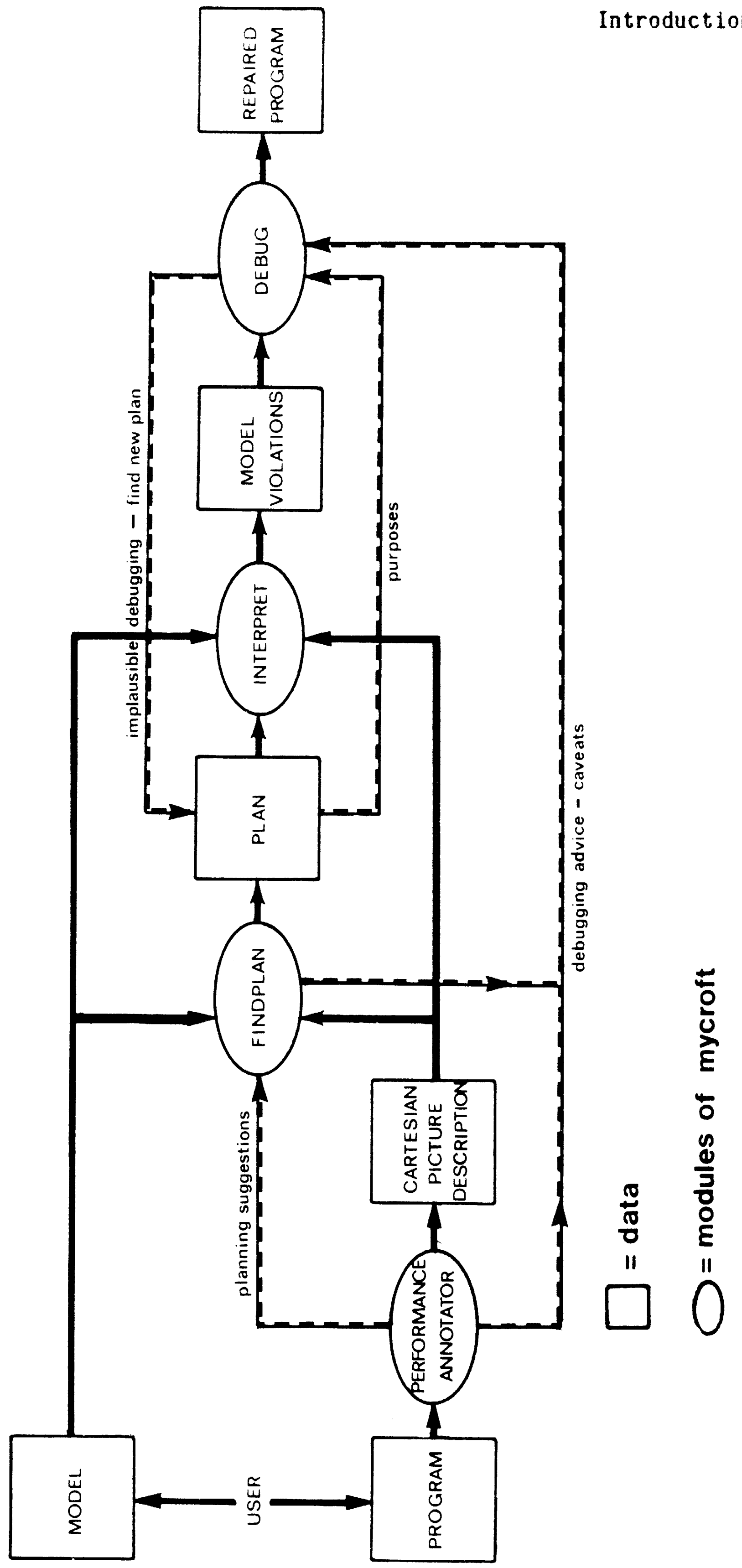


FIGURE 1.11

CHAPTER 2 -- PLANNING

2.1 INTRODUCTION

Picture models describe the intended picture: the turtle program does the actual drawing. Plans serve as a bridge between the two by indicating the problem solving strategy for achieving the model. Plans are a necessary stage in translating a model into an actual program and are vital knowledge for a debugging system. The abstract structure of a plan can supply important suggestions about the underlying causes of bugs. In this chapter, planning knowledge about linearity, preparation, interrupts, global effects and repetition is discussed. Later chapters consider the problems of finding the plan and debugging the program using the plan. A vocabulary for talking about the structure of a procedure is introduced which is useful for understanding both the design and debugging of programs.

2.2 A PLANNING VOCABULARY

A main-step is defined as the code required to achieve a particular sub-goal (sub-picture). A preparatory-step consists of code needed to setup, cleanup or interface main-steps. Thus, from this point of view, a program is understood as a sequence of main-steps and preparatory-steps. A similar point of view is found in [Sussman 1973]. The plan consists of the purposes linking main- and preparatory-steps to the model: in the turtle world, the purpose of main-steps is to accomplish (draw) parts of the model; and the purpose of preparatory-steps is to properly setup or cleanup the turtle state between main-steps or, perhaps, to retrace over some previous vector.

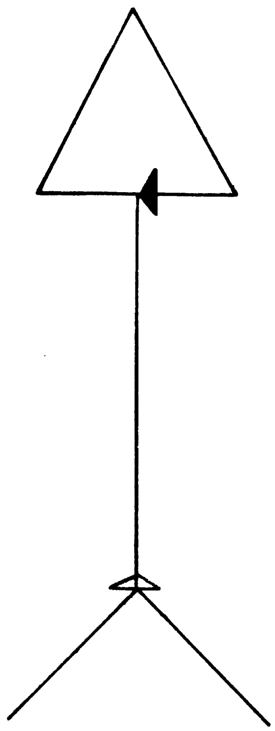
A modular main-step is a sequence of contiguous code intended to accomplish a particular goal. This is as opposed to an interrupted main-step whose code is scattered in pieces throughout the program. In NAPOLEON, the main-steps for the legs, arms and head are modular; however, the body represents an interrupted main-step due to the insertion of the arms into its midst. The utility of making this distinction is that modular main-steps can often be debugged in private (i.e. by being run independently of the remainder of the procedure) while interrupted main-steps commonly fail because of unforeseen interactions with the interleaved code associated with other steps of the plan.

2.3 LINEAR PLANS

Linearity is an important design strategy for creating programs. It has two stages. The first is to break the task into independent sub-goals and design solutions (modular main-steps) for each. The second is then to combine these main-steps into a single procedure by concatenating them into some sequence, adding (where necessary) preparatory-steps to provide proper interfacing. The virtue of this approach is that it divides the problem into manageable sub-problems. A disadvantage is that occasionally there may be constraints on the design of some main-step which are not recognized when that step is designed independently of the remainder of the problem. Another disadvantage is that linear design can fail to recognize opportunities for sub-routinizing a segment of code useful for accomplishing more than one main-step.

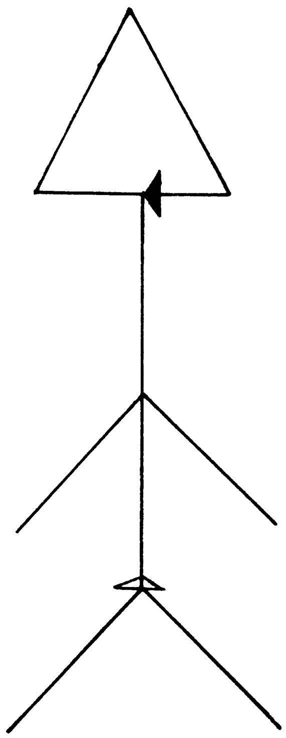
A linear plan will be defined as a plan consisting only of modular main-steps and preparatory steps: a non-linear plan may include interrupted main-steps. The plan of the following stick figure program THINMAN (a subset of NAPOLEON not containing the arm insert) is linear:

```
TO THINMAN
10 VEE          <- (accomplish legs)
20 FORWARD 100  <- (accomplish body)
50 LEFT 90      <- (setup heading for head)
60 TRICORN      <- (accomplish head)
END
```



THINMAN

FIGURE 2.1



Picture of NAPOLEON - turtle starts at HOME

FIGURE 2.2

The concept of linearity provides the basis for both program writing and debugging techniques. With respect to program-writing, constructing procedures using linear simplifications is briefly explored in the final section of this chapter. With respect to debugging, a "linear" approach to correcting bugs is the first technique that DEBUG applies and it is described in depth in the next chapter. It is mentioned here, however, to provide a preview of the imperative use of "linearity" for correcting programs. The first goal in linear debugging is to fix each main-step independently so that the code satisfies all intended properties of the model part being accomplished. Following this, the main-steps are treated as inviolate and relations between model parts are fixed by debugging preparatory-steps. This is not the only debugging technique available to the system, but it is a valuable one for (1) ordering the sequence in which the violations are repaired and (2) limiting the initial search for the repair-point in the program at which the edit for each violation should be made.

2.4 INSERTIONS

In programming, an interrupt is a break in normal processing for the purpose of servicing a surprise. Interrupts represent an important type of plan: they are a necessary problem solving strategy when a process must deal with unpredictable events. Typical situations where interrupts prove useful include servicing a dynamic display, arbitrating the conflicting demands of a time-sharing system, and recovering from certain types of errors. Interrupts can sometimes be used to recover from illegal computations caused by undefined procedures, unbound variables, an incorrect number of inputs or an undefined tag. (These facilities are available in MACLISP [Moon 1973]). The difficulty in anticipating such problems makes interrupts particularly useful. The appropriate correction can be made and the computation recommenced (providing no unrecoverable side effects have occurred). In the real world, biological creatures may use an interrupt style of processing to deal with dangers of their environment such as predators.

A very simple type of interrupt is one in which the program associated with the interrupt is performed for its side effects and is <u>state-transparent</u>, i.e. the machine is restored to its pre-interrupt state before ordinary processing is resumed. As a result, the main process never notices the interruption. In the turtle world, an analogous type of organization is that of an <u>inserted</u> main-step (insertion). It naturally arises when the turtle, while accomplishing one part of a model (the interrupted main-step), assumes an appropriate entry state for another part (the insertion). An obvious planning strategy is to insert a sub-procedure at the desired point in the execution of the interrupted main-step. Often, the insertion will be state-transparent: for turtles, this is achieved by restoring the heading, position and pen state. In the stick-figure example described in chapter 1, the insertion of the arms into the body by statement 30 of NAPOLEON is an example of a position- and pen- but not heading-transparent insertion. Recall that debugging was accomplished by adding a cleanup step to the responsible sub-procedure (VEE) that insured heading transparency.

```
TO NAPOLEON             <- (accomplish man)
10 VEE                  <- (accomplish legs)
20 FORWARD 100          <- (accomplish (piece 1 body))
30 VEE                  <- (insert arms body)
40 FORWARD 100          <- (accomplish (piece 2 body))
50 LEFT 90              <- (setup heading)
60 TRICORN              <- (accomplish head)
END
```

Insertions do not share all of the properties of interrupts. For example, the insertion always occurs at a fixed point in the program rather than at some arbitrary and unpredictable point in time. Nor does the insertion alter the state of the main process as happens in an error handler. However, if one focuses on the planning process by which the user's code was written, then the insertion, as an intervention in accomplishing a main-step, does have the flavor of an interrupt.

Since a plan is relative to the choice of goals, it is important to observe that the same program may represent different plans depending on how the model is expressed. The "parts" constitute the sub-goals for achieving the model and the plan is relative to this division of the picture into parts. For example, if the model for MAN described the figure by:

```
PARTS LEGS, BODY1, ARMS, BODY2, HEAD
```

then statement 30 of NAPOLEON would represent a main-step for the arms rather than an insertion and the design of the program would be linear.

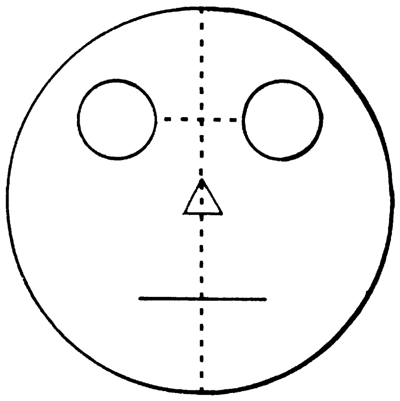
Understanding insertions plays a role in debugging. This occurs through the creation of <u>caveat</u> comments by the plan-finder that warn the debugger of suspicious code. In particular, if FINDPLAN observes an insertion that is not transparent, then a caveat is generated. This occurs, for example, during the analysis of VEE:

```
30 VEE  <- (caveat findplan (not (rotation-transparent insert))).
```

The non-transparent insertion may have been intended. The user's program may be preparing for the next main-step within the insertion. Hence, FINDPLAN does not immediately attempt to correct the anomalous code. The code is corrected only if subsequent debugging of some model violation confirms the caveat. The importance of this advice is that in subsequent debugging, there will often be many possible corrections for a particular model violation. The caveat is used to increase the plausibility of those edits that eliminate FINDPLAN's complaint. In this way, the abstract form of the plan helps to guide debugging.

2.5 SKELETONS

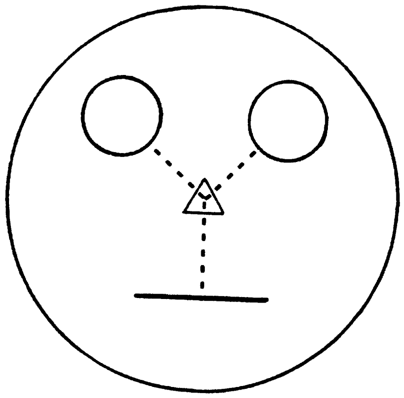
Another type of insertion plan is one wherein all parts are inserted with respect to a single picture object, the skeleton. This object may be itself a model part, an invisible line or simply a point. The FACE picture of figure 2.3 reveals an "invisible skeleton" in which all of the parts of the face are drawn in relation to the vertical line of symmetry passing through the center of the head.



Picture of FACE drawn in relation to a VERTICAL AXIS

FIGURE 2.3

More than one plan usually exists for achieving a given model. Another insertion plan common for faces is to draw all of the parts in relation to the center of the head. See figure 2.4. Here the skeleton is not a part but simply a point. For this special case where the insertions are done with respect to a particular turtle state, the state is called the local home of the plan. A very powerful debugging aid is to recognize that the turtle is intended to return to the "home" following each main step.



Picture of FACE drawn in
relation to CENTER of HEAD

FIGURE 2.4

It is illustrated in section 4.2 which describes the debugging of such a face program.

In an extended system, more complex forms of insertions would have to be understood. A "recursive-snowflake" is built upon the plan that successively smaller triangles are recursively inserted into the sides of their bigger brothers. (The semi-colon commentary shown in the following programs is provided for the reader's benefit and is not generated by the system.)

```
TO SNOW :S :N   ;accomplish snowflake.  See figure 2.5.
10 SIDE :S :N   ;accomplish (side 1 snowflake)
20 RIGHT 120    ;accomplish (rotation 1 snowflake)
30 SIDE :S :N   ;accomplish (side 2 snowflake)
40 RIGHT 120    ;accomplish (rotation 2 snowflake)
50 SIDE :S :N   ;accomplish (side 3 snowflake)
END
```

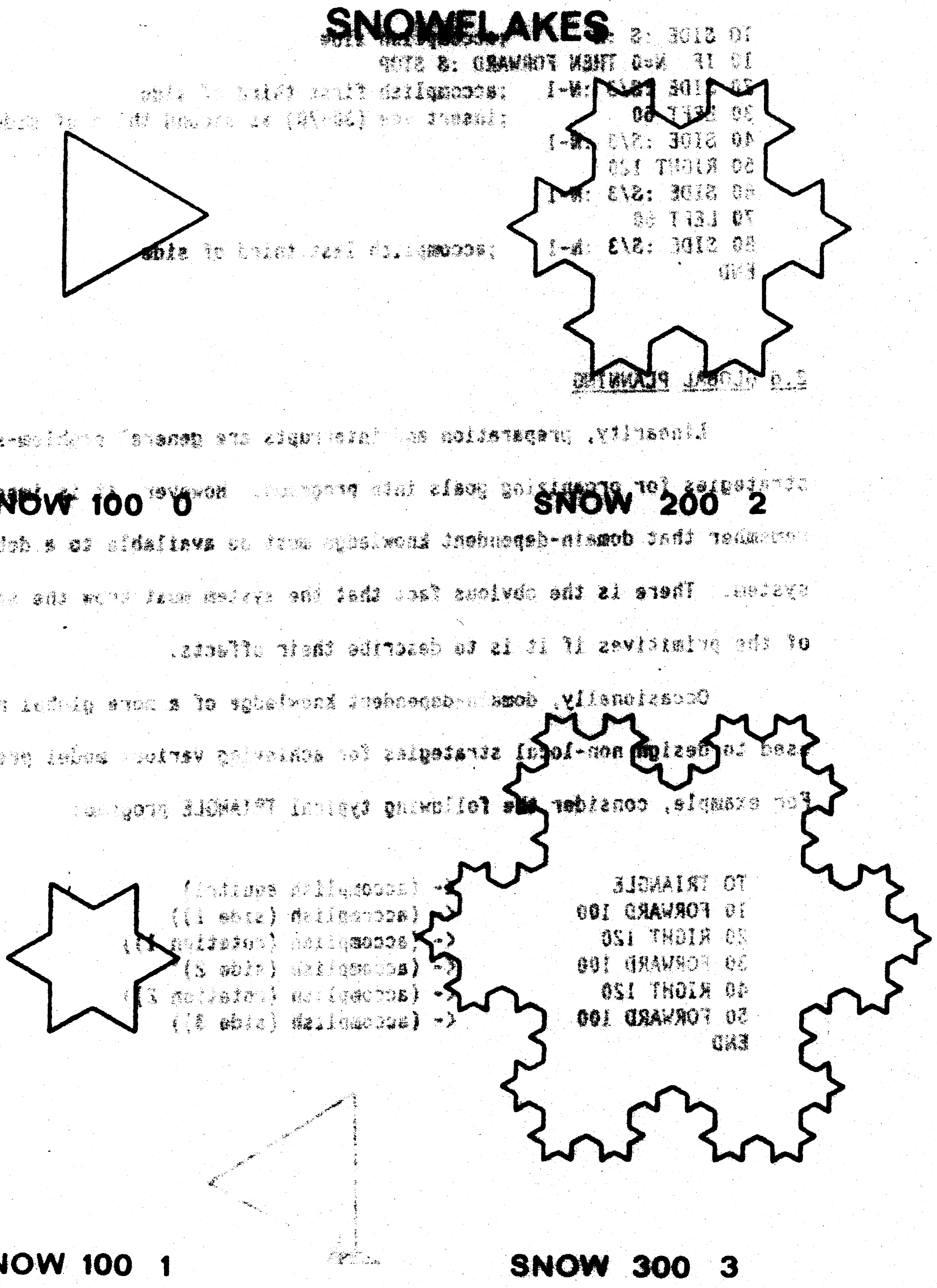


FIGURE 2.5

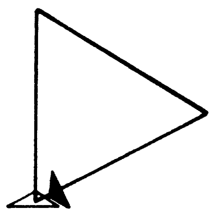
```
TO SIDE :S :N       ;accomplish side
10 IF :N=0 THEN FORWARD :S STOP
20 SIDE :S/3 :N-1   ;accomplish first third of side
30 LEFT 60          ;insert vee (30-70) as second third of side
40 SIDE :S/3 :N-1
50 RIGHT 120
60 SIDE :S/3 :N-1
70 LEFT 60
80 SIDE :S/3 :N-1    ;accomplish last third of side
END
```

2.6 GLOBAL PLANNING

Linearity, preparation and interrupts are general problem-solving strategies for organizing goals into programs. However, it is important to remember that domain-dependent knowledge must be available to a debugging system. There is the obvious fact that the system must know the semantics of the primitives if it is to describe their effects.

Occasionally, domain-dependent knowledge of a more global nature is used to design non-local strategies for achieving various model predicates. For example, consider the following typical TRIANGLE program:

```
TO TRIANGLE          <- (accomplish equitri)
10 FORWARD 100       <- (accomplish (side 1))
20 RIGHT 120         <- (accomplish (rotation 1))
30 FORWARD 100       <- (accomplish (side 2))
40 RIGHT 120         <- (accomplish (rotation 2))
50 FORWARD 100       <- (accomplish (side 3))
END
```



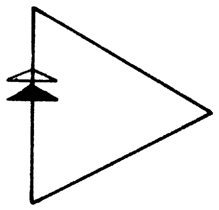
Picture of TRIANGLE

FIGURE 2.6

The responsibility for the fact that (side 1) connects to (side 3) cannot be assigned to any local piece of code. Rather, the closure of the figure is due to the geometric theorem that a broken-line formed from n equal line segments interspersed by equal turtle rotations of 360/n degrees will form a regular polygon.

Another example of global planning in the turtle world is the completion of a side through the collinearity of connecting vectors drawn at different times. An example is the manner in which (SIDE 1) of the corrected TRICORN procedure is completed.

```
TO TRICORN           ;corrected version
                     <- (accomplish equitri)
10 FORWARD 50        <- (accomplish (piece 1 (side 1)))
20 RIGHT 120         <- (accomplish (rotation 1))
30 FORWARD 100       <- (accomplish (side 2))
40 RIGHT 120         <- (accomplish (rotation 2))
50 FORWARD 100       <- (accomplish (side 3))
60 RIGHT 120         <- (accomplish (rotation 3))
70 FORWARD 50        <- (accomplish (piece 2 (side 1)))
END
```

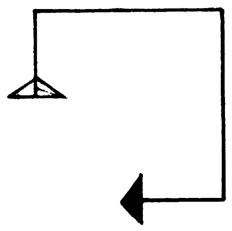


Picture drawn by corrected TRICORN

FIGURE 2.7

In this case, the regular polygon theorem mentioned above is again used to justify that the two pieces of (side 1) are connected. In addition, the fact that the two vectors will have the same heading (and hence be collinear) is based upon the sum of the rotations occurring between them being 0 (mod 360). (An important simplifying characteristic of the turtle semantics is that each primitive affects only one component of the state; hence, the occurrence of the FORWARD instructions in TRICORN is irrelevant to the fact that the two pieces of (side 1) will be parallel.)

Typical bugs of programs based upon global plans are not satisfying the domain theorem used to justify the global effect or applying the wrong theorem. This was the case in the original TRICORN procedure in which all of the rotations were erroneously 90 degrees (figure 2.8). To debug



Picture drawn by TRICORN

FIGURE 2.8

TRICORN, the geometric fact that the sum of the external rotations of a regular polygon equals 360 degrees must be known; or, if this general theorem is not known, then the system must be informed that each rotation of an equilateral triangle must be 120 degrees. (This was, in fact, the case in the EQUITRI model.) The conclusion to be drawn is that powerful debugging systems cannot be based solely on problem-independent techniques but must include mechanisms for utilizing different types of knowledge -- some very specific to the application area. For this reason, the debugger of chapter 3 has access to imperative semantics for geometric primitives and specific theorems about turtle geometry in addition to general debugging strategies for correcting programs.

2.7 OPEN-CODING

In earlier examples, the plan has been indicated by "<-" commentary which describes the purpose of each statement. This representation is not adequate for purposes which extend over more than one statement. For example, in the following "open-coded" program TREE3, the TOP and TRUNK are accomplished by non-subroutinized code. The code for TOP is not even contiguous.

```
MODEL TREE
M1 PARTS TOP TRUNK
M2 LINE TRUNK
M3 EQUITRI TOP
M4 VERTICAL TRUNK
M5 COMPLETELY-BELOW TRUNK TOP
M6 CONNECTED TOP TRUNK
M7 HORIZONTAL (BOTTOM (SIDE TOP))
END
```

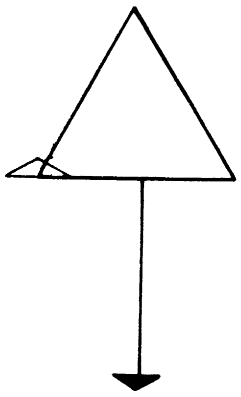


FIGURE 2.9

```
TO TREE3          ;Semi-colon commentary is provided for
                  ;the reader's benefit and is not generated
                  ;by the system.
 5 RIGHT 30
10 FORWARD 100    ;first side of the triangle.
20 RIGHT 120
30 FORWARD 100
40 RIGHT 120
50 FORWARD 50     ;first half of the third side
60 LEFT 90
70 FORWARD 100    ;insert of trunk
80 BACK 100
90 RIGHT 90
100 FORWARD 50    ;completion of the third side.
END
```

To define such plans, PURPOSE statements are used which explicitly mention the code associated with the planning statement.

```
(PURPOSE (TREE3 5) (SETUP HEADING FOR TOP)
(PURPOSE (TREE3 10-50, 100) (ACCOMPLISH TOP))
(PURPOSE (TREE3 60-90) (INSERT TRUNK TOP))
```

Open-coded sequences are segments of non-subroutinized code responsible for a given purpose. This is reflected in the code appearing as a group in the <code> part of a PURPOSE statement. These PURPOSE statements are MYCROFT's internal representation of the plan. The "<-" planning commentary represents a pretty-print of these purposes. The "<-" representation for the plan for TREE3 is shown below, with open-coded sequences displayed as sub-routines named by the sub-model being accomplished.

```
TO TREE3         <- (accomplish tree)
 5 RIGHT 30      <- (setup heading)
10 FORWARD 100   <- (accomplish (side 1 top)
20 RIGHT 120     <- (accomplish (rotation 1 top)
30 FORWARD 100   <- (accomplish (side 2 top)
40 RIGHT 120     <- (accomplish (rotation 2 top)
50 FORWARD 50    <- (accomplish (piece 1 (side 3 top)))
(60-90) LINE     <- (open-coded insert for trunk)
100 FORWARD 50   <- (accomplish (piece 2 (side 3 top)))
END


TO LINE          <- (accomplish line)
60 LEFT 90       <- (setup trunk)
70 FORWARD 100   <- (accomplish trunk)
80 BACK 100      <- (cleanup position)
90 RIGHT 90      <- (cleanup heading)
END
```

Treating programs as labeled statements allows a more flexible approach to describing the program than the notion that commentary can exist solely at the interfaces between sub-procedures. This latter point of view is found in Hewitt's Actor formalism for computation [Hewitt 1973]. It is inadequate, however, to describe the evolution of programs, insertion type planning structures, or protections.

The identification of open-coded sequences with a common purpose as sub-routines is important for subsequent debugging. In TREE3, sub-routine names are generated for the triangle code and for the insert of the TRUNK. Sub-routinization is not solely done for the economy of a single representation for frequently used code. Perhaps more important is its use for achieving conceptual modularity. It greatly facilitates debugging and planning. It allows private debugging techniques (section 3.5). For example, identifying the lines of the triangle "sub-routine" allows the system to consider whether the triangle would be successful if the insert for the TRUNK was removed. If so, debugging can be focused on the interactions of the two segments and rely on the assumption of first-order success, i.e. that the main-steps accomplish their purposes.

2.8 PURPOSE STATEMENTS

The following table summarizes the syntax and vocabulary for the planning assertions.

PURPOSE <code> <explanation>

Examples of <code>:

(TREE3 5)	= statement 5 of TREE3.
(TREE3 60-90)	= statements 60 through 90 inclusive. Referred to as an open-coded sequence.
(TREE3 10-50 100)	= statements 10 to 50, inclusive, and statement 100.

Examples of <explanation>:

(ACCOMPLISH TRUNK)	= Draw the trunk.
(INSERT TRUNK TOP)	= Accomplish the trunk by a state transparent sub-procedure inside the code for TOP.

(ACCOMPLISH (PIECE i (SIDE 1 TRIANGLE))) = Accomplish piece i of side 1 of the triangle. The remainder will be accomplished by another piece of code. Depending on the intervening program, this indicates either a global or an insertion plan.

(SETUP HEADING FOR TRUNK) = Set up the state -- in this case the heading -- in preparation for accomplishing the trunk.

(SETUP HEADING SUCH-THAT (VERTICAL TRUNK)) = Set up the state -- in this case the heading -- in preparation for the next step. The "such-that" indicates the model statement to be satisfied by the preparation, and is optional.

(RETRACE P1) = Preparatory step in which the <code> is accomplished in such a way that it overlaps a visible sub-picture P, making itself "invisible".

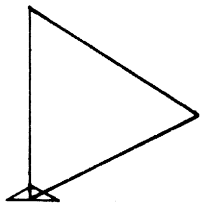
Wherever possible, these assertions will be pretty-printed as "<-" commentary for ease of reading, although the internal form manipulated by the system is as given above.

2.9 ROUND PLANS

The remainder of this chapter proposes extensions to the MYCROFT system. This section discusses programs containing simple loops and the next considers the program-writing problem.

The first step in extending the planning vocabulary beyond those plans used in fixed instruction procedures would be to develop a description of round-structured programs. These are programs in which a basic round is repeated some number of times. Round plans must be introduced which include a description of simple control patterns such as an arithmetic counter, increment function and conditional for controlling the number of repetitions. An example of a round program for a triangle accompanied with planning commentary is:

```
TO ITERATIVE.TRIANGLE           <- (accomplish triangle)
10 MAKE "SIDES" 0               <- (initialize counter :sides)
20 IF :SIDES = 3 THEN STOP      <- (exit condition)
30 FORWARD 100                  <- (round (accomplish side))
40 RIGHT 120                    <- (round (accomplish rotation))
50 MAKE "SIDES" :SIDES+1        <- (increment counter)
60 GO 20                        <- (loop boundary)
END
```



Picture of TRIANGLE – turtle starts and ends at HOME

FIGURE 2.10

The basic round draws a side and a rotation of the triangle and is repeated three times. See figure 2.10.

The existence of such primitives as DO, WHILE and MAP in various high-level languages like LISP and ALGOL indicates the utility of round-structured plans. The important elements of counter, exit condition and round are made specific slots in a "repetition" primitive rather than requiring the user to define a control structure each time as in the previous ITERATIVE.TRIANGLE program. For example, in terms of the FORTRAN DO primitive, ITERATIVE.TRIANGLE becomes:

```
TO ITERATIVE.TRIANGLE
10 DO 30 I=1,3  ;Repeat from here to statement 30 three times.
20 FORWARD 100
30 RIGHT 120
END
```

This is part of the "structured-programming" approach to simplifying debugging [Dyjkstra 1972]. Clarifying types of plans is clearly relevant to structured-programming in that it can suggest higher level primitives to incorporate in the language for the purpose of making these planning structures explicit and less subject to bugs.

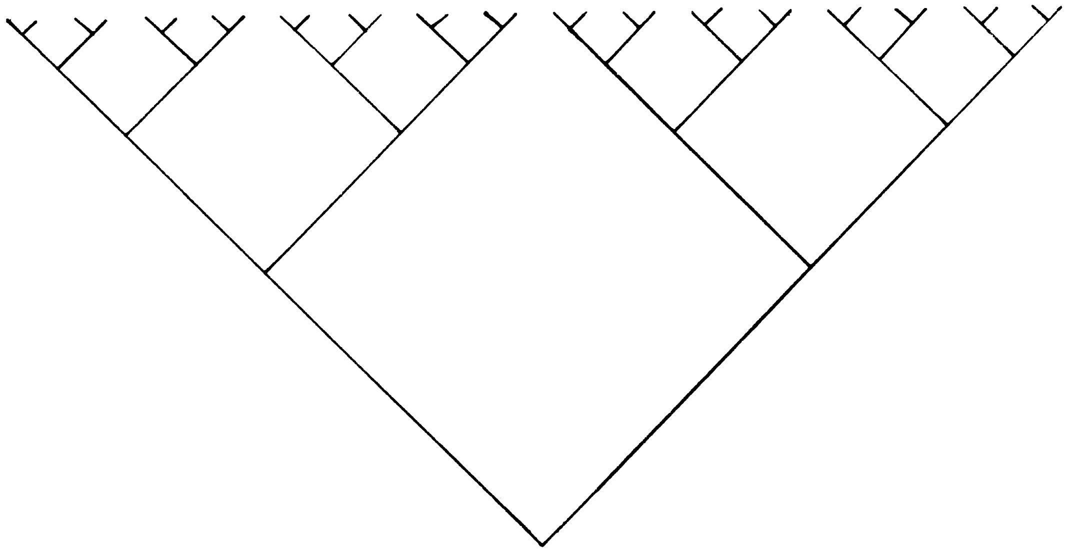
Round plans are particularly appropriate when the picture is described in terms of a generic or typical element [Winston 1970]. An obvious situation where this might occur is in the description of an equilateral triangle. A generic rotation of 120 degrees and generic side of some fixed length provides not only an economical description of the figure but also suggests the use of a round-structured program to accomplish instances of the generic part. EQUITRI, the equilateral triangle model given in chapter 1, uses this typical element description:

```
MODEL EQUITRI           ;generic model
M1 PARTS (SIDE 3) (ROTATION 3)
M2 FOR-EACH SIDE (= (LENGTH SIDE) 100)
M3 FOR-EACH ROTATION (= (DEGREES ROTATION) 120)
M4 RING CONNECTED SIDE
END
```

(See appendix B for details of this extension to the picture language for describing "typical elements".) A round plan for the ITERATIVE.TRIANGLE procedure given above would associate statement 20 with accomplishing the generic side and statement 30 with the generic rotation.

A round plan can be implemented as either a recursive or iterative procedure. In more sophisticated programming, iteration and recursion are differentiated by such criteria as power, efficiency and modularity. Tags can have overlapping scopes: recursion can occur in the interior rather than only at the end of the program: procedures can be co-recursive. For example, the following pair of co-recursive routines draws the binary tree of figure 2.11. An additional complexity is that :BRANCH is not passed explictly as an input to SUBTREE but instead is passed indirectly as a free variable.

```
TO BINARY.TREE :BRANCH
10 IF :BRANCH < 20 THEN STOP
20 SUBTREE 45
30 SUBTREE -45
END

TO SUBTREE :ANGLE
10 LEFT :ANGLE
20 FORWARD :BRANCH
30 RIGHT :ANGLE
40 BINARY.TREE :BRANCH/2
50 LEFT :ANGLE
60 BACK :BRANCH
70 RIGHT :ANGLE
END
```



BINARY TREE

FIGURE 2.11

However, for elementary round programs, this complexity is prohibited.

In an extended system, it is possible that the user might associate a model in which the parts are explicitly named with a recursive or iterative program. The planning formalism must be designed to allow proper binding of parts to code in such situations. An example is the following recursive triangle program and explicit model.

```
TO TRIANGLE                   MODEL TRIANGLE
10 FORWARD 100                M1 PARTS S1 S2 S3 R1 R2 R3
20 RIGHT 120                  M2 (EQUAL S1 S2 S3)
30 TRIANGLE                   M3 (= (DEGREES R1) 120)
END                           M4 (CONNECTED S1 S2)
                                      .
                                      .
```

To match code with model parts properly, the following syntax is proposed:

```
(PURPOSE (TRIANGLE 10 (ROUND 1)) (ACCOMPLISH S1)).
(PURPOSE (TRIANGLE 10 (ROUND 2)) (ACCOMPLISH S2)).
(PURPOSE (TRIANGLE 10 (ROUND 3)) (ACCOMPLISH S3)).
```

The round number becomes a part of the <code> designation.

The system is not knowledgeable about more complex control structures. For example, it is not prepared to analyze interrupts which do not return the world to the pre-interrupt state as happens with insertions. Such interrupts are appropriate for real time applications such as processing touch-turtle impacts. Nor is it familiar with backtracking, the PLANNER control mechanism for handling failure [Hewitt 1972, Sussman 1970]. Being ignorant of such control structures, their corresponding plans and typical bugs, the system would be unable to understand or debug programs using them. Future systems might be extended to handle such issues.

2.10 PROGRAM WRITING

This final section concludes with a brief excursion into Program-Writing to illustrate the intimate relation between planning, debugging and program-writing skills. Given the model, knowledge of plans can be used to guide a top-down planning process which begins by specifying the purposes of the main-steps. The main-steps are then treated as sub-goals, with code for each main-step being written by recursing the system and following the same line of attack, i.e. writing a program for the main-step on the basis of the sub-model describing the part to be accomplished. The preparatory-steps of the template are then debugged, details being added to the interfaces to make the predicates describing relations between main-steps true.

> To write a program:
> 1. Pick some subset of the model and build a plan;
> 2. Write a procedure for this plan;
> 3. Write sub-procedures for main-steps;
> 4. Debug the procedure to satisfy the remaining model statements.

Ultimately, the model parts are described as primitive vectors or rotations for which the imperative semantics for the turtle primitives apply.

The level of detail needed in a model to debug is often less than is required to write a program. The reason for this is that the debugger has access to the code which supplies additional constraints while the program-writer has nothing but the model to guide its analysis. An interesting extension would be to supply a sketch with the model (or even instead of the model) in order to provide quantitative values for the size of the parts and their relative positions.

The first step in this very simplified model of program-writing is to build a linear super-procedure by assigning a sub-procedure call for each model part. Although it is important not to get enmeshed in details in writing the initial program, some principles of good design can be used to provide an ordering for these main-steps. From a heuristic standpoint, this represents the notion that a well-chosen sequence can simplify later debugging of the preparatory-steps. In the turtle world, this is done by ordering the super-procedure on the basis of transitive sequences of such predicates as CONNECTED, ABOVE and INSIDE.

LINEAR PLAN

1. Accomplish model parts by modular main-steps.
2. Concatenate main-steps into a procedure. Prefer an ordering suggested by transitive sequences of such predicates as CONNECTED, ABOVE, and INSIDE occuring in the model.
3. Design preparatory-steps between each pair of main-steps for satisfying relations between the model parts accomplished by these main-steps.

For the MAN model, this planning paradigm would produce the procedures SKELETON and LIMBS on the assumption that the system has already associated VEE with the model V and TRICORN with the model EQUITRI.

```
TO SKELETON                                   TO LIMBS
10 VEE            <- (accomplish legs)        10 VEE     <- (accomplish legs)
20 FORWARD :BODY  <- (accomplish body)        30 VEE     <- (accomplish arms)
60 TRICORN        <- (accomplish head)        60 TRICORN <- (accomplish head)
END                                           END
```

The length of the body is not known so the variable :BODY is used. It is up to the user to supply additional detail in order to determine the proper value.

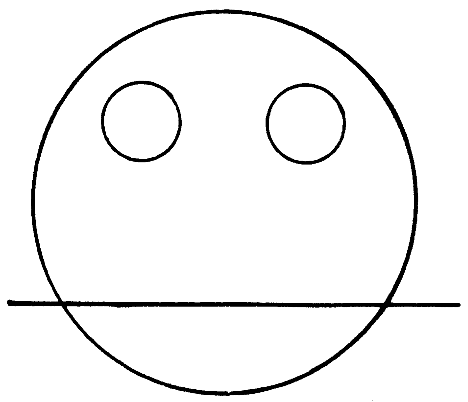
The next step is to accomplish, where possible, any unexplained parts by inserts. SKELETON becomes closer to NAPOLEON by using the clue that the arms are connected to the body between the head and legs. (This must be deduced from the combination of connection and below constraints.) The result is the insertion of statement 30.

INSERT PLAN	1. If several model parts (limbs) are connected to a common skeleton, then accomplish the skeleton in pieces, inserting the limbs at the appropriate points.
	2. Order limbs by transitive sequences.

Thus, from this point of view, inserts represent an elaboration of linear plans for the purpose of accomplishing parts which do not fit easily into a linear sequence.

These plan-generating criteria first produce a plan for accomplishing the parts. This is a reasonable problem solving technique. It amounts to simplifying the problem to obtain a first approximation to the solution. Achieving this approximate solution would be pointless, however, if it could not be extended to a complete program. Fortunately, this can indeed be done. The remaining model statements describing relations between the parts are accomplished by writing preparatory-steps. This is done by running the simplified precedure and using the debugger to generate the code to be added at the main-step interfaces. For example, in SKELETON, statement 50 LEFT 90 is added to make the HEAD completely above the BODY. Thus, in this view of programming, debugging is a necessary ability. The style of program-writing is to simplify the problem, build an approximate solution, and then debug it until it satisfies all of the constraints.

If debugging produces irreconcilable conflicts, a new plan must be chosen. Sometimes the solution is to reorder the main-steps. At other times, conflicts between a main-step and a preparation may require the preparation to be moved to some earlier point in the procedure. Both of these debugging techniques are discussed in Sussman [1973]. In some situations, the independent design of the main-steps may be at fault and no reordering will be successful. This is the case for BIG.MOUTH (figure 2.12). The mouth is too large for the face. The important fact in such



BIG.MOUTH
A non-linear bug in a face

FIGURE 2.12

cases is that the way in which the original linear plan fails often provides guidance as to which alternative plan to choose.

In some circumstances, a program-writer should not begin with a linear plan. This arises, for instance, if the picture can be described in terms of a generic element. A typical example is the description of the sides and rotations in the EQUITRI model given earlier. The obvious strategy here would be to use a Round Plan.

ROUND PLAN

1. If the parts are generic, write a sub-procedure (the round) which accomplishes the typical element.

2. Repeat the round by the number of instances required by the PARTS declaration.

The program-writer would then recurse and write the sub-procedure for the basic round in terms of the generic description of the typical element.

This has been a very brief sketch of a difficult problem. However, its purpose has been only to provide some intuition for the relationship between planning, debugging, and program writing. Program-Writing using <u>debugging</u> <u>with</u> <u>no</u> <u>planning</u> lacks any global insight into the structure of the problem. <u>Planning</u> <u>with</u> <u>no</u> <u>debugging</u> entails the difficult burden that the plan be perfect the first time. <u>Planning</u> <u>with</u> <u>debugging</u> simplifies the programming task by first searching for a incomplete schema and then trying to fill in details and correct oversimplifications until the entire goal is met.

CHAPTER 3 -- MODEL-DRIVEN DEBUGGING

3.1 MODEL VIOLATIONS

Access to models and plans allows a new dimension in error detection. For most programming systems, including LISP and LOGO, the only errors recognized automatically are syntactic--those which cause illegal parsing of the input--and semantic--those which cause illegal computations. A typical parsing error in both LOGO and LISP is the occurrence of unmatched parentheses. Illegal computations are caused by attempts to make illegal memory references, to execute undefined procedures, to jump to undefined tags or to pass the wrong type or number of arguments to sub-procedures. These errors are recognizable without requiring any special knowledge of intent.

MYCROFT is designed to debug model violations. These are recognized by comparing the output of a syntactically and semantically correct turtle program (i.e. a program that is able to run to completion without requesting any illegal computations) to the description of intent provided by its picture model, using the plan to bind sub-pictures to model parts. The result is a list of violated model predicates. The program is considered correct when all of these violations have been explained and eliminated.

> The distinction between semantic errors and model violations is a difference of manifestation and not a profound criterion for discovering the underlying cause of the failure. The underlying cause of semantic errors may be conceptual mistakes in planning, forgotten prerequisites, or an inadequate understanding of the interactions of competing goals. Similarly, model violations may be simply due to a mistake in understanding the performance of a programming primitive, failing to read in the proper files or even a trivial syntactic mis-typing. For simplicity, however, the debugging discussion will limit itself to the correction of model violations.

Correcting model violations is accomplished by using two types of procedural knowledge. The first are general debugging strategies for repairing programs and the second are specific directions for fixing particular geometric and logical predicates. Because overall guidance is derived from the model, I shall call this type of analysis <u>model-driven debugging</u>.

In this chapter, I shall assume the existence of a plan linking the model to the program. The problem of finding the plan is postponed until chapter 7. The next section provides an example of the debugger's performance. This is followed by an analysis of the knowledge required to debug turtle programs. Chapter 4 provides more examples and chapter 5 concludes the debugging discussion with an overview that considers alternatives to model-driven debugging.

3.2 DEBUGGING TREE1

3.2.1 Bugs

This section presents a scenario in which MYCROFT successfully debugs a tree procedure. The program TREE1 exhibits three errors.

```
TO TREE1          ;version 1
10 TRIANGLE       <- (accomplish top)
20 RIGHT 50       <- (setup heading)
30 FORWARD 50     <- (setup position)
40 RIGHT 50       <- (setup heading)
50 FORWARD 100    <- (accomplish trunk)
END
```

The rotation of line 20 is a "local preparation" error in that it fails to cause the position setup (line 30) for the trunk to retrace a side of the triangle. An unexpected runtime environment causes an improper orientation for the top of the tree. Finally, the rotation of line 40 which

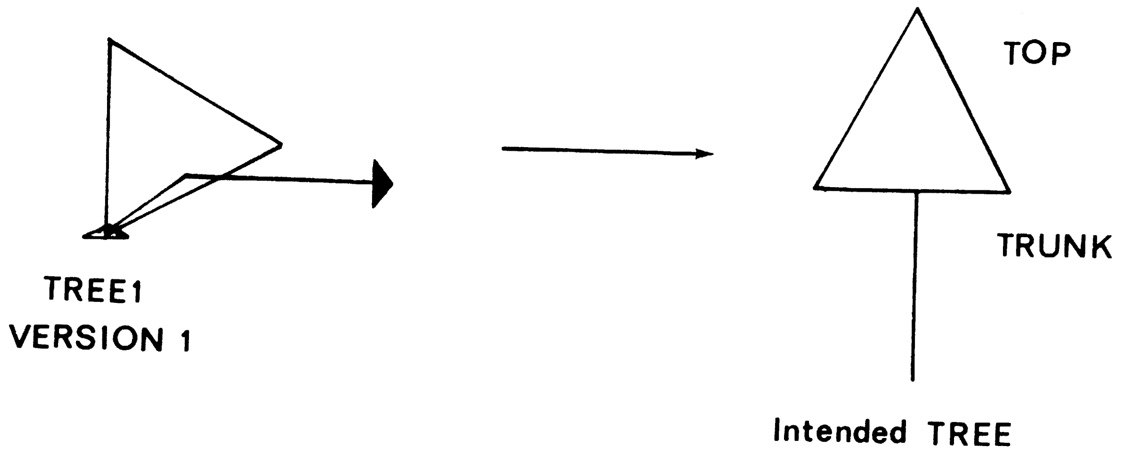


FIGURE 3.1

establishes the orientation of the trunk is incorrect. MYCROFT successfully debugs all of these problems.

3.2.2 Interpretation

Recall that MYCROFT requires, in addition to the user's programs, a model of the intended picture. Many such predicate descriptions are possible. This scenario will be based on the following typical TREE description.

```
MODEL TREE
M1 PARTS TOP TRUNK
M2 LINE TRUNK
M3 TRIANGLE TOP
M4 VERTICAL TRUNK
M5 COMPLETELY-BELOW TRUNK TOP
M6 CONNECTED TOP TRUNK
M7 HORIZONTAL (BOTTOM (SIDE TOP))
END
```

To evaluate the success of the program with respect to model, two ingredients are necessary.

1. The actual performance of the program must be documented. For the turtle world, this is done by describing the picture

produced by the turtle in Cartesian terms. This annotation is generated by a careful evaluation of the program in which imperative semantics for the turtle primitives generate descriptive assertions in a database. This is described in chapter 6.

2. The plan must be available to describe the purpose of the code in terms of the model. The plan is generated by the system if the user fails to supply it. Section 7.7 describes the process of finding the plan for TREE1. For this debugging scenario, assume that the plan (as shown above) has been correctly deduced.

With knowledge of the program's performance and purpose, INTERPRET can generate a list of those model predicates which are violated. (See figure 1.11 for the place of INTERPRET in the system flowchart.) This list of violations is the input to DEBUG. For TREE1, these violations are:

```
(NOT (HORIZONTAL (SIDE TOP)))
(NOT (INVISIBLE (INTERFACE BETWEEN TOP AND TRUNK)))
(NOT (CONNECTED TOP TRUNK (VIA ENDPOINTS)))
(NOT (VERTICAL TRUNK))
(NOT (COMPLETELY-BELOW TRUNK TOP))
```

3.2.3 Ordering the Violations

MYCROFT now applies ordering criteria for deciding the sequence in which the violations will be debugged. This is important to minimize fruitless corrections which are only undone by the repair for a subsequent error. One typical criterion is to debug properties before relations. This is based on the problem-solving strategy of solving sub-problems independently, when possible, before considering the difficulty of properly fitting them together.

Another ordering criterion is to repair those violations wherein the debugger is reasonably sure that the cause lies directly in the responsible code and is not due to interactions with earlier parts of the procedure. This criterion is useful for deciding to debug CONNECTED before

BELOW. The rationale for this is that connections are topological and independent of the frame of reference. Hence, the repair-point must be in the code between the TOP and the TRUNK. This is a consequence of the fact that for fixed-instruction turtle programs, the initial environment determines only the frame of reference and does not affect topological properties. Fixed-instruction turtle programs draw rigid bodies. (The implications for debugging of this rigid body property are discussed further in section 3.7.2.) For BELOW, the repair-point is less constrained and may even be due to the initial environment in which the TREE1 procedure is executed. (This criterion is not useful for further sorting the violated properties in TREE1, as all of these violations are probably due to some bug in the code prior to the main-step which accomplished the violated part.)

Finally, if the violations are not yet completely ordered, the subsets are debugged according to the temporal sequence in which the model parts were accomplished. (Further details regarding these ordering criteria and their rationale are developed in section 3.4.) For TREE1, the result of this ordering is the following sequence of violations:

```
;Violations of Properties
        (NOT (HORIZONTAL (SIDE TOP)))
        (NOT (INVISIBLE (INTERFACE BETWEEN TOP AND TRUNK)))
        (NOT (VERTICAL TRUNK))

;Violations of Relations
        (NOT (CONNECTED TOP TRUNK (VIA ENDPOINTS)))
        (NOT (COMPLETELY-BELOW TRUNK TOP))
```

3.2.4 Overview of the Debugger's Performance

The ordered violations are corrected in sequence. This involves choosing for each violation the culpable code and then making the appropriate edits to the turtle program. Figure 3.2 shows the changes to the picture as the program is edited.

The following table indicates the monitor's conclusions regarding the abstract cause of each violation.

```
(NOT (INVISIBLE (INTERFACE BETWEEN TOP AND TRUNK))
                      <=> Local Preparation Error in statement 20.

(NOT (CONNECTED TOP TRUNK (VIA ENDPOINTS)))
                      <=> Same cause as the visible interface.

(NOT (HORIZONTAL (SIDE TOP)))
                      <=> Unexpected Runtime Environment

(NOT (COMPLETELY-BELOW TRUNK TOP))
                      <=> Same cause as (NOT (HORIZONTAL (SIDE TOP)))

(NOT (VERTICAL TRUNK)) <=> Local Preparation Error in STATEMENT 40.
```

The end result is the picture and program shown below. The underlined commentary are additions to the plan inserted by the debugger.

```
TO TREE1         ;version 4
 5 RIGHT 30     <- (setup heading such-that (horizontal (side top)))
10 TRIANGLE     <- (accomplish top)
20 RIGHT 60     <- (setup heading such-that
                     (retrace (interface statement 30) over (side 3 top))
30 FORWARD 50   <- (retrace (side 3 top))
40 RIGHT 90     <- (setup heading such-that (vertical trunk))
50 FORWARD 100  <- (accomplish trunk)
END
```

3.2.5 Making the Bottom Side of the Top Horizontal

The imperative semantics for HORIZONTAL guides the debugging process in fixing (HORIZONTAL (SIDE TOP)). The definition of HORIZONTAL is the disjunction:

(OR (= (DIRECTION V) 90) (= (DIRECTION V) 270)).

To make a side horizontal, the difference between its direction and 90 or

DEBUGGING TREE1

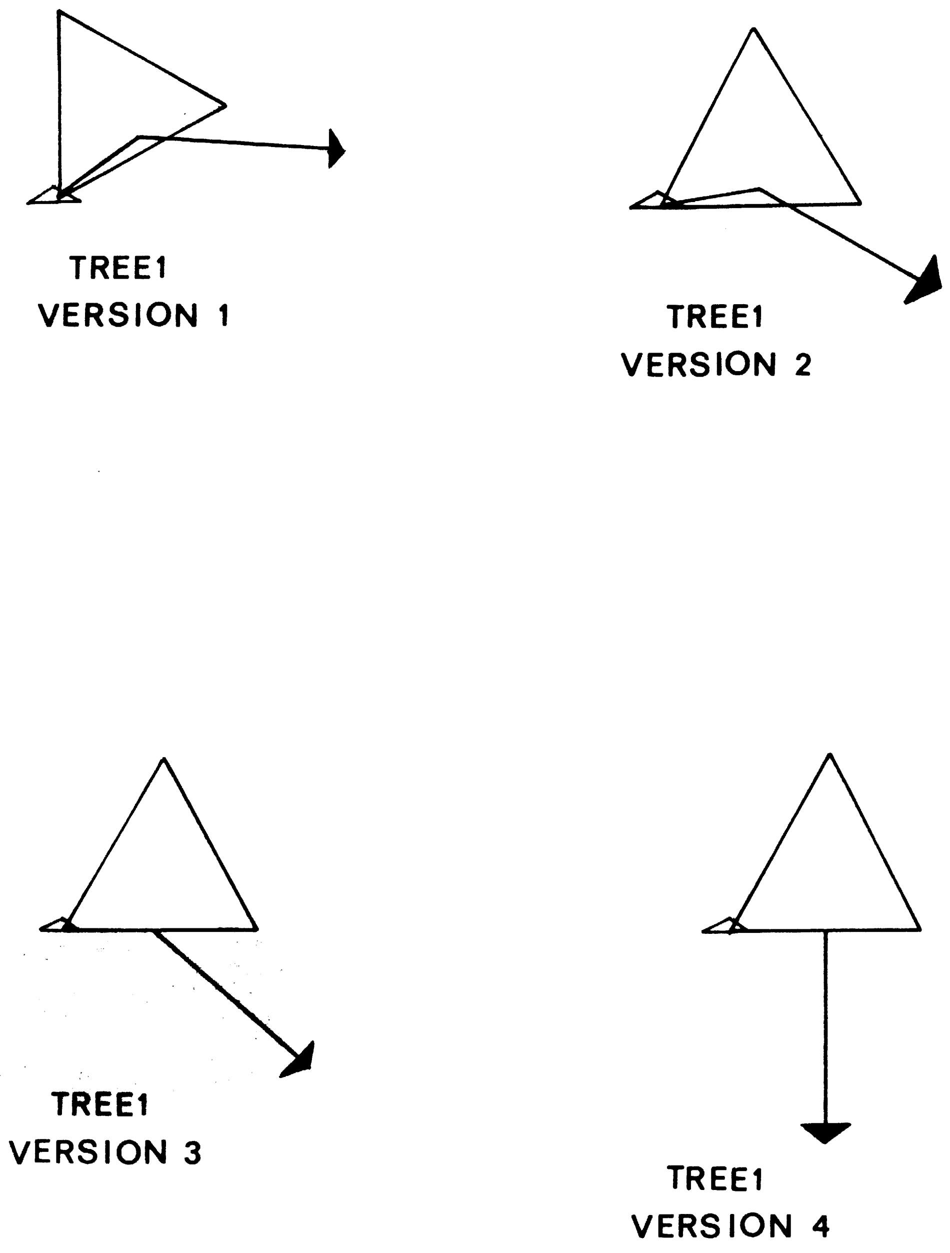


FIGURE 3.2

270 is determined. The debugger must consider where in the procedure the corrective rotation is to be added, i.e. the location of the repair-point. Placing the rotation inside the TRIANGLE sub-procedure after statement 10 has bad side effects in that it destroys the triangle shape. Hence, the rotation is associated with the entry state to the triangle code. This still leaves two possibilities: (1) insert the rotation directly into the beginning of the TRIANGLE sub-procedure or (2) add the rotation as statement 5 to TREE1. Because the debugger is reluctant to modify the code for a part which already satisfies its sub-model (in this case EQUITRI), the edit is made to TREE1 directly. (Section 3.6 elaborates the criteria for when the edit is inserted into a sub-procedure versus when it is inserted directly into the super-procedure.) The result is an edit which modifies the entry state of TREE1 and consequently the error is classified as an Unexpected Runtime Environment.

A choice exists with respect to which side of the TOP is to be made horizontal. The user has not asserted directly:

(HORIZONTAL (SIDE 3 TOP)).

Choosing (side 1) or (side 2), however, leads to bad side effects. The TRUNK is currently PARTLY-BELOW the TOP but this is undone by making either of these two sides horizontal. Thus, the system prefers to make (side 3) horizontal. The result is TREE1, version 2.

```
TO TREE1    ;version 2
 5 RIGHT 30 <- (setup heading such-that
                   (horizontal (side 3 top)))
            <- (assume (enter TREE1 line 5) (= :heading 0))
            <- (protect :heading to TRIANGLE statement 60)
10 TRIANGLE
20 RIGHT 50
30 FORWARD 50
40 RIGHT 50
50 FORWARD 100
END
```

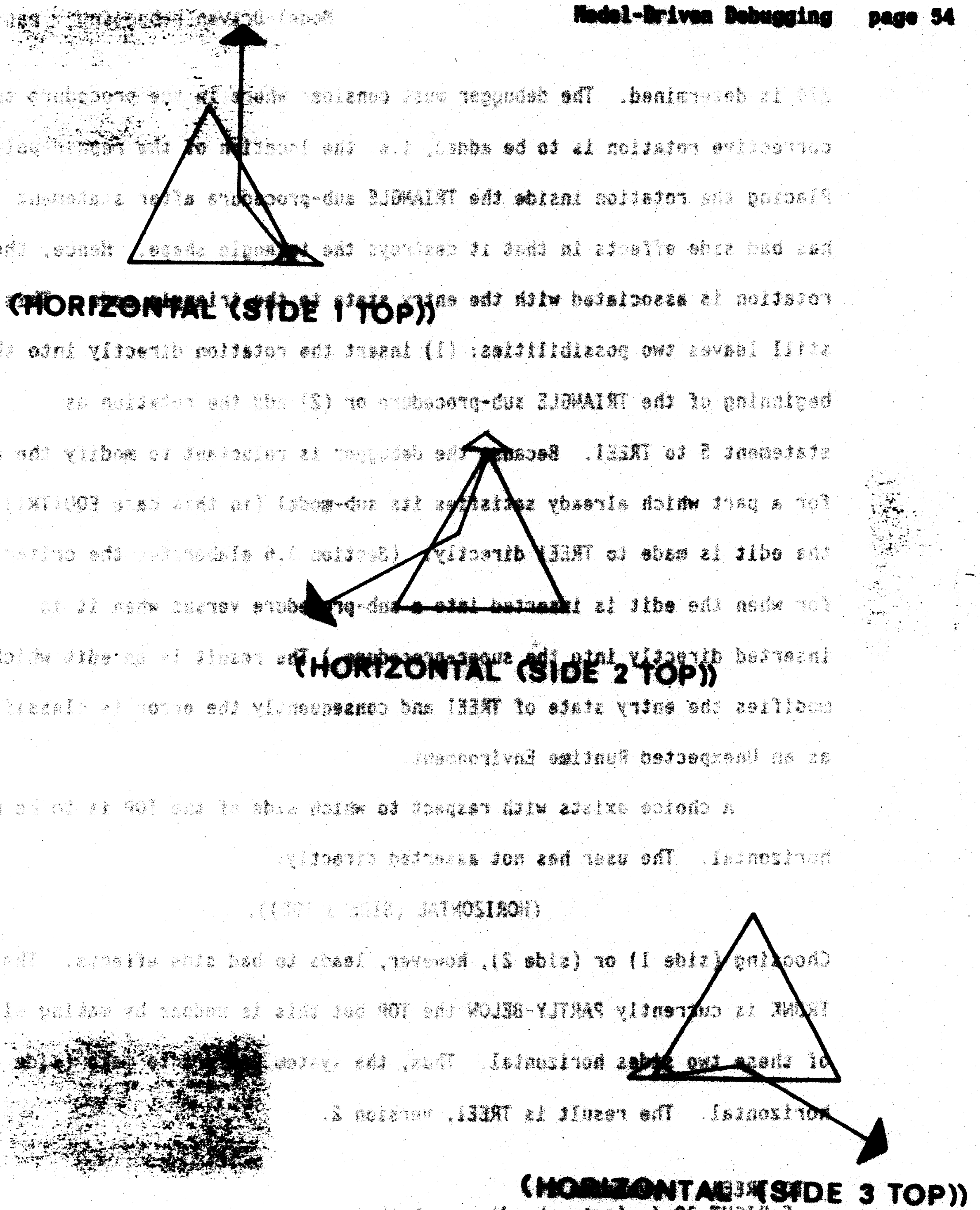

(HORIZONTAL (SIDE 1 TOP))
(HORIZONTAL (SIDE 2 TOP))
(HORIZONTAL (SIDE 3 TOP))

FIGURE 3.3

3.2.6 Making the Interface Invisible

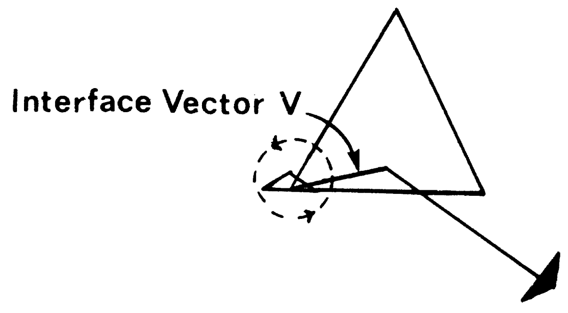
The interface consists of the vector V drawn by statement 30. The repair strategy is guided by the imperative semantics for INVISIBLE. This consists of the following disjunction:

```
(OR (= (PENSTATE V) :UP) (RETRACE V W))
```

The first disjunct indicates that V will be invisible if the pen is up during its creation. The alternative strategy is for V to retrace the path of some visible vector W.

Debugging invisibility by making the first disjunct true is accomplished by inserting PENUP into the code prior to statement 30. Inserting the PENUP before the TRIANGLE of statement 10 is prevented by the bad side effect of making the TOP invisible. A PENUP insertion immediately prior to statement 30 corrects the interface but has the bad side effect of making the TRUNK invisible. This is not fatal since a PENDOWN might then be added prior to drawing the TRUNK. Such would be the repair strategy selected if retracing were unsuccessful. However, retracing is not only possible, it results in beneficial side effects and is therefore preferred.

The alternative debugging strategy of retracing requires a choice of the picture vector W over which the vector drawn by statement 30 is to lie. The debugger prefers to satisfy the imperative semantics by making minimal changes to the user's code. Hence, the first possibility is to restrict the correction to altering the direction of V by inserting or editing a rotation and refraining from also adding additional vector instructions. This has the effect of rotating V around the sub-picture drawn prior to the point in the program at which the rotation is being inserted. Figure 3.4 shows the effect of inserting a rotation immediately prior to statement 30 in TREE1. Other possibilities based on translating



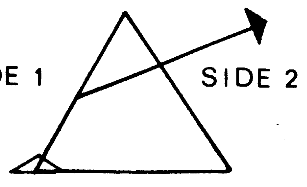
Picture of TREE1 -
turtle starts at HOME

FIGURE 3.4

as well as rotating V are not pursued since the rotation change has the desired effect of causing V to overlap a visible vector.

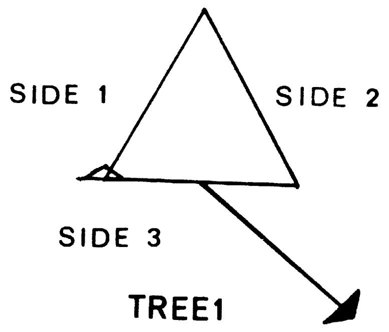
The first candidate for the repair-point which is analyzed is the one immediately preceding the location in the procedure which the error became manifest. This is the rotation at statement 20. V can be made invisible by modifying the input to the RIGHT command in statement 20. Indeed there are two possible candidates for the retrace: (SIDE 1 TOP) and (SIDE 3 TOP).

Rotating V to coincide with (SIDE 1 TOP) is rejected since it produces the additional violation of the TRUNK being inside the TOP. (This is a violation since COMPLETELY-BELOW requires that the TRUNK be outside the TOP.) A beneficial side effect occurs if V is rotated to coincide with (SIDE 3 TOP) of the triangle, namely that the connection between the TOP and the TRUNK is moved to an endpoint. Hence this is the preferred debugging strategy and statement 20 is edited to be RIGHT 60. The error is classified as a Local Preparation Error and TREE1, version 3, is produced.



Interface rotated to overlap
(SIDE 1 TOP)

FIGURE 3.5



TREE1
VERSION 3
Interface rotated to overlap
(SIDE 3 TOP)

FIGURE 3.6

```
TO TREE1          ;version 3, figure 3.6
 5 RIGHT 30
10 TRIANGLE
20 RIGHT 60       <- (setup heading such-that
                       (retrace (interface statement 30) (side 3 top)))
                  <- (protect :heading from triangle statement 60
                                   to treel statement 30)
30 FORWARD 50     <- (retrace (side 3 top))
40 RIGHT 50
50 FORWARD 100
END
```

3.2.7 Making the Trunk Vertical

Debugging (VERTICAL TRUNK) is very similar to debugging the previous horizontal violation. (VERTICAL V) is defined by:

(OR (= (DIRECTION V) 0) (= (DIRECTION V) 180)).

From an imperative standpoint, the first step is to compute the required rotation needed to establish the proper orientation of the TRUNK. This is a rotation of 40 or 220 degrees. Again by the rigid body nature of fixed-instruction turtle pictures, this rotation can be inserted anywhere into the code prior to statement 50 and have the proper effect upon the orientation of the TRUNK. The system explicitly considers initially only the interfaces at statements 5, 20 and 40. Again the interior of the triangle procedure is considered inviolate since it already satisfies its sub-model EQUITRI. (The heuristic of editing interfaces and treating main-steps as inviolate once they satisfy their sub-model is based on the linear notion that interactions occur only at explicit interfaces. This constraint on the debugging process is useful in limiting the number of repair points considered but not always successful. Its limitations are discussed further in section 3.6).

The initial setup at statement 5 is already constrained by the edit to make the bottom side of the triangle horizontal. This is reflected in the protection commentary. Similarly, the interface at statement 20 is constrained by the previous edit to establish the invisibility of statement 30. Hence, the preferred location for the repair-point is at statement 40.

Recall that there are two possible headings, 0 and 180, for making the TRUNK vertical. The correction to make the TRUNK vertical by establishing a direction of 0 degrees (figure 3.7) is rejected since it causes the TRUNK to overlap the TOP. On the other hand, no bad effects occur from increasing this rotation by 30 degrees and giving the TRUNK a direction of 180 (figure 3.8). This is the preferred correction and the result is that statement 40 is edited to be "RIGHT 90".

```
TO TREE1    ;version 4, figure 3.8.
 5 RIGHT 30
10 TRIANGLE
20 RIGHT 60
30 FORWARD 50
40 RIGHT 90 <- (setup heading such-that (vertical trunk))
            <- (assume (enter TREE1 statement 40) (= :heading 90))
            <- (protect :heading through TREE1 statement 50)
50 FORWARD 100
END
```

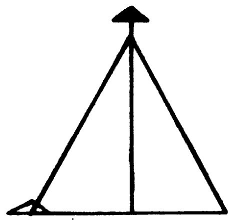
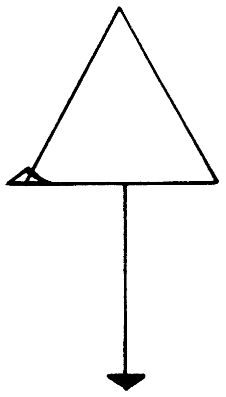

DIRECTION OF TRUNK = 0°

FIGURE 3.7


DIRECTION OF TRUNK = 180°

TREE1

VERSION 4

FIGURE 3.8

Making the TRUNK vertical has the beneficial side effect of establishing the most restrictive type of BELOW relation between the TOP and the TRUNK. The TRUNK is not only completely below the TOP, even the centers of gravity are aligned. Hence, version 4 is the final program and all of the violations are corrected. It is shown below complete with the additional commentary added by the debugger.

```
TO TREE1        ;version 4
 5 RIGHT 30     <- (setup heading such-that (horizontal (side 3 top)))
                <- (assume (enter TREE1 statement 5) (= :heading 0))
                <- (protect :heading to TRIANGLE statement 60)
10 TRIANGLE     <- (accomplish top)
20 RIGHT 60     <- (setup heading such-that
                      (retrace (interface statement 30) (side 3 top)))
                <- (protect :heading from triangle statement 60
                                  to treel statement 30)
30 FORWARD 50 <- (retrace (side 3 top))
40 RIGHT 90     <- (setup heading such-that (vertical trunk))
                <- (assume (enter TREE1 statement 40) (= :heading 90))
                <- (protect :heading through TREE1 statement 50)
50 FORWARD 100     <- (accomplish trunk)
END
```

3.3 DEBUGGING AS SEARCH

A debugging strategy is a sequence of editing commands whose effect is to modify the program so that it satisfies its model. There are generally multiple debugging strategies for correcting a given set of violations. These debugging strategies arise from choice of the location at which the corrections are to be made as well as of the exact meaning that the user intended.

To clarify the issues which arise in selecting the best debugging sequence, it is useful to conceptualize the problem in terms of a search metaphor. The space is that of all possible debugging strategies for correcting the program. Each node is a set of model violations: the origin of the space is the initial set of violations. An arc is an edit which which leads to a node containing the new (and presumably fewer) set of violations which are produced by the modified code. Branching occurs for each possible patch for correcting a violation. A path through the space constitutes a series of edits that transforms the program to an acceptable form. For example, there are at least four possibilities for correcting the visible interface violation of TREE1,

1. Add PENUP immediately before statement 30;
2. Add PENUP before statement 10;
3. Edit statement 20 such that statement 30 retraces (side 1 top);
4. Edit statement 20 such that statement 30 retraces (side 3 top).

Recognizing the existence of multiple possibilities for correcting a program, it is appropriate to ask what knowledge is used to:

1. Choose the next model violation to be debugged?
2. Generate the possible corrections for that violation?
3. Choose the most plausible correction?

The following sections answer these questions. Ordering criteria are introduced for choosing the sequence in which the violations are debugged. A linear approach curtails the number of possible edit points which are initially considered. The imperative semantics of the model predicates are used to generate possible corrections. Plausibility criteria are designed for selecting among alternative debugging strategies.

3.4 ORDERING MULTIPLE VIOLATIONS

Multiple bugs are difficult to fix. Guidelines are required to order the sequence in which the violations are debugged. These guidelines reflect an understanding of dependency relationships between violations, thereby serving to minimize the unfortunate occurrence of a correction undoing previous repairs or introducing new violations. The ordering is done on the basis of preferring to repair:

(1) bugs in properties of model parts before bugs in relations between model parts;

(2) bugs in intrinsic properties (or relations) before bugs in extrinsic properties (or relations);

and (3) bugs occurring earliest in the temporal sequence of execution.

The following paragraphs describe these criteria and explain their rationale.

3.4.1 Debug Properties Before Relations

The system debugs violations of properties of model parts before repairing violations of relations between model parts. This is based on the important heuristic of first having a successful theory of the parts before attempting an explanation of their interactions. This is more than good style. The behavior of the interfaces is designed relative to the entry-exit states of the code for the main-steps accomplishing the parts.

To determine the specific state changes to be made at an interface, the performance of adjacent main-steps must be established. Thus the code for sub-pictures must be fixed prior to deciding on the proper edits to the preparatory-steps.

Properties of individual model parts are described either by one-place model primitives (e.g. VERTICAL, HORIZONTAL, LINE) or user-defined models (e.g. MAN, V, EQUITRI). Properties are first-order constraints on model parts because they are independent of relations between model parts.

The second-order description is built from relations beteen model parts. The most common are such predicates as ABOVE, BELOW, and CONNECTED. These are debugged only after the properties are corrected.

3.4.2 Debug Intrinsic Before Extrinsic Predicates

The idea behind the next ordering criterion is to estimate the range of possible locations in the program at which the repair might be made for each violation. Let the scope of a violation be the code between the repair-point and the manifestation-point. The heuristic is then to fix those violations of most-limited scope first; both because they are easiest and because of dependency relationships.

For a property (P M), M a model part, the manifestation-point is the location in the program at which M is completed and the truth of the statement (P M) can be evaluated. The repair-point is the location in the program at which the edit is eventually made to correct the violation. For a relation (R M N), the manifestation-point is the location in the program at which both M and N have been completed and the relation R can be evaluated.

The heuristic of fixing violations of limited-scope first would be pointless if there were no way to estimate the scope of a violation before entering into the details of debugging. However, this is not the case. One method for estimating the scope of a violation is to know whether the property of relation is intrinsic to the responsible code.

A property (P M) is intrinsic to the code for M if it is independent of preceding code and entirely due to the main-step for M. Similarly, the relation (R M N) is intrinsic if it is independent of code preceding M, assuming that M is achieved before N. Repair is simplified by fixing intrinsic predicates before extrinsic ones since (1) for intrinsic violations, the possible repair-points are easier to find since they cannot occur prior to the code for M, and (2) the proper corrections for extrinsic predicates depend upon the the code being intrinsically correct.

In the world of turtle geometry, intrinsic errors are distinguished by being independent of the frame of reference: they cannot be corrected by translating or rotating the picture. This is because in the simplified environment of fixed-instruction turtle programs, code groups draw rigid bodies. The initial interface of a code group has the effect of establishing the origin and orientation of the sub-picture but does not affect the local relations among vectors. Topological predicates (invariant under transformations that preserve connectivity) and geometric predicates (invariant under translation and rotation) are independent of the frame of reference and therefore yield intrinsic violations. Bugs in the following model primitives are always intrinsic to the code group to which they refer: OVERLAP, INSIDE, OUTSIDE, PARALLEL and CONNECTED.

Extrinsic errors are those affected by the initial environment in which the code group is executed. The initial environment consists of the bindings of the turtle state variables -- :HEADING, :POSITION and :PEN.

These variables control the orientation, origin and visibility of the sub-picture as well as its relation to previously drawn parts of the picture. Model predicates which depend on the initial state are VERTICAL, HORIZONTAL, BELOW, and ABOVE.

Debugging intrinsic violations first tends to establish the proper connections at interfaces. Debugging extrinsic relations like ABOVE and BELOW then becomes simply a matter of establishing the proper heading at interfaces.

In the turtle world, the distinction between intrinsic and extrinsic predicates is particularly easy to make; however, it remains a useful debugging distinction in other domains. If a property of a program is due to some local data structure (such as a bound variable) or local control structure (such as a loop) and is independent of the preceding code, then it is intrinsic and worth debugging in private before extrinsic properties (whose causes are less easy to isolate) are repaired.

3.4.3 NAPOLEON's Violations

The following list of violations for NAPOLEON is ordered by the above criteria:

```
;Violations of Properties
  ;Intrinsic Predicate
    (NOT (EQUITRI TRICORN))

  ;Extrinsic Predicate
    (NOT (LINE BODY))

;Violations of Relations
  ;Temporal Order -- {legs, arms} accomplished before {arms, head}.
    (NOT (BELOW LEGS ARMS))
    (NOT (BELOW ARMS HEAD))
```

3.5 PRIVATE DEBUGGING

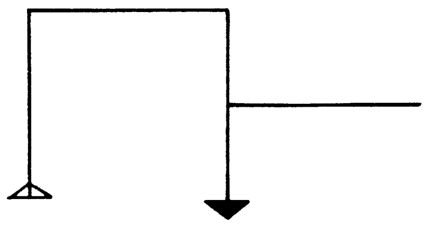
For each violation, DEBUG must find the proper repair-point in the program at which to insert the correction. Of course, the debugger knows that the repair-point cannot follow the code for the parts mentioned in the violation but this is hardly a sufficient constraint. Consequently, DEBUG uses two heuristics--Private and Linear Debugging--to limit the possible locations for the correction. Private debugging is described in this section: linear debugging in the next.

An initial heuristic for constraining the possible repair-points for a violated property is to limit consideration to the code directly responsible for the model part in question. This is done by running the responsible code independently of the larger procedure of which it is a part. Specifically, the responsible code is executed with the turtle started at the entry state. The violated properties will be manifested in this private environment if the main-step is modular. However, if there is intervening code, i.e. the main-step is interrupted, then the linear assumption that the cause is intrinsic to the responsible code and not due to interactions may be wrong.

If the violation is manifest, the code group is then debugged in this simplified context, free of the effects of the remainder of the original program. Private debugging is used to repair the three incorrect rotations of TRICORN. There are no complications when the edited sub-procedure is rejoined to the NAPOLEON super-procedure.

The following program is a bugged version of TREE3, the tree program discussed in section 2.7, in which the trunk is inserted into the bottom side of the triangle. The bugs are underlined.

```
TO BUGGY.TREE3
10 FORWARD 100  <- (accomplish (side 1 top))
20 RT 90        <- (accomplish (rotation 1 top))
30 FORWARD 100  <- (accomplish (side 2 top))
40 RT 90        <- (accomplish (rotation 2 top))
50 FORWARD 50   <- (accomplish (piece 1 (side 3 top)))
60 LEFT 90      <- (setup heading for trunk)
70 FORWARD 100  <- (insert trunk (side 3 top))
80 BACK 100     <- (cleanup position)
90 RIGHT 90     <- (cleanup heading)
100 FORWARD 50  <- (accomplish (piece 2 (side 3 top)))
END
```



BUGGY.TREE3

FIGURE 3.9

Debugging in private results in statements 10-50 and 100 being treated as a sub-procedure. The rotations are properly edited to be 120 degrees, Only then is the program considered in its entirety, with possible errors due to the insert being analyzed. This is an example in which debugging is greatly simplified by knowing the plan: without knowledge of the insert, the debugger would flounder in considering lines 60 to 90 as possibly part of the triangle.

The relationship between the picture drawn in private and in public is simple for fixed-instruction turtle programs since the picture is a rigid body and only its orientation and origin is affected by the initial environment. For more complex programs, difficulty occurs in finding a representative private environment and further research is necessary. This is similar to the problem of diagram generation in geometry theorem proving and to the problem of case analysis in automatic program verification.

The private repair may make assumptions about the entry state to the code. If this happens, it will be reflected in ASSUME comments regarding the entry state to the main-step. When run again in the real context, any conflicts between assumptions made in private about the initial environment and the actual entry state are themselves debugged. This is accomplished by adding code to accomplish the assumptions in the super-procedure or, if this proves impossible without causing additional violations, backtracking and attempting an alternative correction in private.

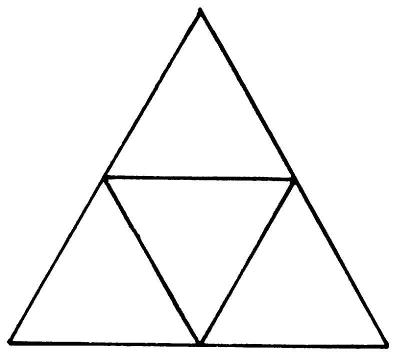
An example of this would occur if the model for NAPOLEON had declared that the body must be vertical. Debugging the body (statements 20 and 40) in private would result in the assumption being generated that the entry heading must be 0 or 180 degrees. The code for the body is then reconsidered in the context of the NAPOLEON super-procedure. The actual entry state to statement 20 does not have :HEADING equal to 0 or 180 degrees. Consequently, the debugger now attempts to add a rotation at some preceding point in the program to achieve this entry state. This addition will most likely occur immediately prior to statement 20 or, perhaps, as the initial setup to the NAPOLEON program. The debugger chooses whether to prefer 0 or 180, and at which repair-point, on the basis of side effects, minimal change to the user's program and planning caveats. This set of plausibility criteria is described in section 3.8.

The system also checks for bad side-effects on code following the edited sub-group due to a new exit state for the edited code. A cleanup step may be needed to eliminate undesirable consequences of the private repairs. The modified main-step may violate protection or assumption commentary generated by other edits. If so, the standard practice is to either (1) modify the offended edit in light of the new structure for the main-step or (2) backtrack and correcting the main-step in private in some alternative way. Section 3.10 provides details on the protection mechanism.

Occasionally, when the code is run in private, the violation does not occur. This happens because the main-step is not modular and the violation is due to code appearing between pieces of an interrupted main-step. Private debugging remains useful, however, because it clearly indicates that the cause of the error is in the intervening code. For NAPOLEON, (NOT (LINE BODY)) is an example: the body when run in private is indeed a line. The bug is in the effect of the inserted VEE on the heading of the second vector.

Private debugging is also used to correct intrinsic violations of relations. Recall that the definition of an intrinsic relation is that it is entirely due to the code between the model parts mentioned in the relation. Hence, the repair-point must occur there. The same precautions required when the code is rejoined to the super-procedure--i.e. satisfying assumptions, and possibly cleaning up--must be taken. Outside the turtle world where it may not be so easy to decide if a relation is intrinsic, private debugging can still be attempted. Just as for properties, if the violation does not appear in private, then it is known that it is not intrinsic and the system can look for causes in preceding code.

For fixed-instruction programs, there cannot be conflicts of inputs or free variables. The influence of the outside world on the sub-procedure is completely confined to the initial turtle state -- position, heading, pen. In more complex programs, bugs not amenable to "private debugging" become more common due to such reasons as conflicts over free and bound variables. Another situation where private debugging is inappropriate is in fixing relationships accomplished globally. An extreme case of this occurs in the following "5 triangles out of 2". The global method for accomplishing the five triangles does not correspond to any local code segment which can be considered privately.



5 TRIANGLES FROM 2

FIGURE 3.10

3.6 LINEAR DEBUGGING

Linear Debugging is a technique for limiting the possible repair-points for correcting violations of both the intrinsic and extrinsic kind. It is based upon the assumption that DEBUG has already privately repaired the main-steps to satisfy their properties. The linear debugging technique is to consider editing corrections only at preparatory-steps and not internal to the code for the main-steps. Main-steps are treated as inviolate black-boxes: their contents need neither be known nor changed. This is based upon the assumption that the main-steps are independent and that the only corrections necessary to repair relations is to make adjustments at interfaces. This was the technique used to debug (VERTICAL TRUNK) in TREE1. DEBUG limited the search for the proper edit by not considering the addition of a rotation to the interior of the TRIANGLE sub-procedure. Instead, it restricted itself to an analysis of possible corrections at the level of the TREE1 super-procedure.

It is necessary to disobey the linear prohibition against modifying main-steps if the restriction to editing preparatory steps is unsuccessful. This may happen because preparatory edits either fail to eliminate the violation, or succeed in eliminating the violation but conflict with previous edits or introduce new unfixable problems. Linear debugging will fail when the cause of a violation of a relation between two model parts is due to the code for one of the parts. GOOGLY.EYES (figure 3.11) is an example. The bug in the overlap of the eyes and head is not in the interface but in the size of the eyes.

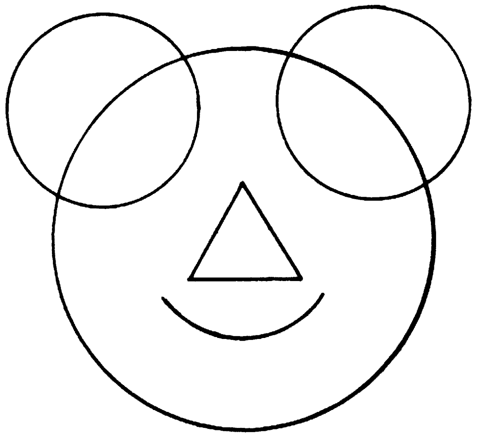

GOOGLY.EYES

FIGURE 3.11

3.7 IMPERATIVE KNOWLEDGE

How is the set of possible edits for repairing a violation generated? The answer lies in the use of procedural knowledge associated with the model primitives which provides direction on how to make the predicate true. The system has imperative knowledge for logical primitives like equality and conjunction as well as for geometric primitives appropriate to the turtle world. This imperative knowledge is represented as FIX programs which are invoked by the particular model predicate being debugged. The output of a FIX program is a sequence of directions for the editor which eliminates the violation.

In the NAPOLEON example, (NOT (EQUITRI TRICORN)) is a violation of a user-model. Such violations are fixed by recursive entry to the debugger and analyzing the code for the model in private. Such recursion ultimately reduces the debugging to fixing violations of model primitives.

3.7.1 Imperative Knowledge for Geometric Primitives

The following discussion describes in a simplified way the imperative knowledge associated with several of the model primitives. Appendix D describes these semantics in greater detail. Let X and Y be vectors and assume that X is accomplished before Y.

(LINE X Y) <=> (AND (PARALLEL X Y) (CONNECTED X Y))

The imperative semantics for AND directs the debugger to establish the two relations of PARALLEL and CONNECTED. These are defined below.

(PARALLEL X Y) <=> (= (DIRECTION A) (DIRECTION B) (MOD 180))

The annotator records the DIRECTION of vectors. The repair is to insert rotations between the code for X and the code for Y so that the direction of Y becomes equal to the direction of X (mod 180).

(VERTICAL X) <=> (OR (= (DIRECTION X) 0) (= (DIRECTION X) 180))

Alter preceding rotations so as to make the direction of X 0 or 180.

(CONNECTED X Y)

Choose a connection point on X (P1) and a connection point on Y (P2). The connection point is sometimes specified in the model: for example, the user may have indicated that it should occur (AT (MIDDLE (SIDE ...))). Then compute the vector V from P1 to P2. The edit is to add code for V into an interface between X and Y. This will have the effect of translating Y so that P1 is moved to coincide with P2.

If the exact position is unknown, deduce it from constraints such as preferring to effect the code in minimal ways. This is done by manipulating individually the length and angle inputs to translation and rotation interface steps (occurring between the code for X and the code for Y) and observing if X and Y intersect as a result. Branch in considering alternative allowable connection positions.

(ABOVE X Y) - (similar technique for BELOW, RIGHT-OF, LEFT-OF)

To compute the required correction for a given interface: assume that the figure has already been debugged to be topologically correct--e.g. all of the connections are correct. This implies that the only degree of freedom in interfaces is the heading.

In considering a given interface, find the range of headings which satisfy the predicate. The range is determined by first finding the heading of most restrictive meaning of ABOVE -- CENTERED-ABOVE wherein the center of gravity of X is directly above Y. Then relax this heading to find the maximum range in which less restrictive meanings of the predicate--COMPLETELY-ABOVE and PARTLY-ABOVE--remain true. To select a specific heading to actually edit into the code, choose the value that satisfies the most restrictive meaning of ABOVE. If there is still a range of possible headings, use the average value. Record the range considered in case later debugging results in conflicts and another heading must be chosen.

3.7.2 The Rigid Body Theorem

Fixed-instruction turtle programs draw rigid bodies, i.e. the only effect of the initial runtime environment is to alter the visibility, origin or orientation of the frame of reference. This theorem simplifies the generation of possible repair edits by allowing computation of the required rotation for HORIZONTAL, VERTICAL and PARALLEL to be made only once, independently of the point in the code at which the edit is to be added. This is useful since there are usually many points at which patching the code must be considered to fix these violations.

For example, suppose the side of a triangle is to be made horizontal. The required rotation is computed for the side. However, if the edit is made immediately prior to the code for the side, the triangle shape will be destroyed. The rotation, however, can be added to preceding code, rotating all subsequent vectors the same amount and consequently still making the side horizontal.

In general, if the correction is a rotation of the frame of reference, the edit can be added anywhere prior to the code group to be rotated. If the rotation is to change the relation between two sub-pictures, then it can often occur anywhere in the code occuring between the main-steps which accomplish the sub-pictures.

3.7.3 Imperative Knowledge of Logical Predicates

The general advice for fixing (= (P A) (P B)) is to use the imperative semantics for property P to either make (P A) equal to (P B) or vice versa. For the simple case of fixed-instruction turtle programs, the change is usually made to A or B on the basis of which occurs last. This is preferred because of the rigid body nature of sub-pictures. For example, suppose A occurs before B. Then adding RIGHT :ANGLE before A rotates A but it also rotates B. An opposite rotation must be added after A if B is not to be affected by the first edit. Thus, fixing the sub-picture which occurs first commits the system to two changes of the program. Of course, editing the code before B may also require a cleanup because of bad side effects but this is not inevitable as it is in the first case. This preference is reflected in the general debugging criteria of avoiding conflicts, minimizing change to the user's program and preferring beneficial side effects.

Thus, fixing equality consists of:

(1) General Knowledge: Either A or B can be fixed. Prefer to alter the unprotected element (section 3.10); and

(2) Domain-Dependent Knowledge: Imperative semantics are provided for relating primitives to their effects.

These semantics for primitives are used by the annotator to document the effect of a statement of code, and by the debugger to add the correct code to achieve a desired effect. For example, to alter the direction of a vector, the annotation semantics for FORWARD (section 6.3) indicate that the DIRECTION property of vectors is equal to the current heading. The annotation semantics for RIGHT indicate that :HEADING is incremented by :ANGLE following execution of "RIGHT :ANGLE". The conclusion drawn by the debugger, then, is that either "RIGHT :ANGLE" is needed to fix the direction of B or "RIGHT -:ANGLE" is needed to fix the direction of A, where :ANGLE equals the difference between the desired direction and the actual direction.

To fix (AND C1 C2 ...), correct all of the conjuncts. Order the debugging attack on the basis of the same criteria used to order the initial set of violations, i.e. (1) correct properties of main-steps before correcting relations between main-steps, (2) correct intrinsic before extrinic predicates and (3) debug a given group of conjuncts at the same level (with respect to the preceding criteria) in temporal order.

See appendix D for a description of imperative semantics for other model primitives such as INSIDE, OUTSIDE, OVERLAP, OR, NOT and FOR-EACH.

3.8 DECIDING BETWEEN ALTERNATIVE DEBUGGING STRATEGIES

More than one debugging strategy is usually available to fix a given violation. The strategies differ with respect to their estimate of the repair-point and with respect to the type of correction they apply to fix a given model violation. For example, the imperative semantics for VERTICAL indicate the desired direction but allow the correction to be added into any prior interface. In TREE1, the trunk can be made VERTICAL by adding the appropriate rotation to the procedure as either statement 5, 20 or 40. The preferred debugging strategy is the one that does minimal violence to the user's code, reflects the abstract plan, and fixes the greatest number of violations.

3.8.1 Plausibility on the Basis of Side Effects

The first criterion for judging the success of a particular debugging strategy is an analysis of the side effects of the corrections. The debugging strategy with maximal beneficial side effects is preferred. Beneficial side effects occur by eliminating additional model violations, satisfying planning expectations or eliminating violations of rational form. (The latter is defined in section 3.9 on the state editor.)

> One might ask why an edit might have any beneficial side effects at all. Isn't it more likely to have bad side effects and cause other violations? The answer is that often several violations are caused by the same error in the code. Then one debugging strategy will stand out from its brethren by fixing this error and thereby simultaneously curing several violations.

On the other hand, sometimes a correction causes additional model violations. In this case, either the new violations can themselves be debugged or the debugging strategy must be abandoned. Assumption and protection commentary are used to help in understanding those bad side effects wherein one edit undoes the effect of some other debugging edit. This is discussed in section 3.10. If the bad side effect cannot be eliminated, then the debugging strategy must be rejected. This is the case with a linear debugging of GOOGLY.EYES (figure 3.11). The eyes cannot be brought into the head by shrinking the interface without causing them to overlap the nose. Thus this debugging strategy eliminates one violation (OVERLAP EYE HEAD) only to introduce another (OVERLAP EYE NOSE). The system is forced to consider non-linear debugging and fix the parts themselves.

3.8.2 Plausibility on the Basis of Minimal Change

Another plausibility criterion is that of minimal change to the user's code. A debugging strategy that changes an input is preferred to one that adds lines; and a strategy that adds lines is in turn preferred to one that deletes them. The rationale is that a repairman should make minimal changes to a system. The goal is to fix the program in harmony with the user's intent, not to redesign it. This caution is further justified by the fact that the system does not fully know the programmer's intent or plan. Hence, it must be hesitant to make major revisions to his program.

3.8.3 Plausibility on the Basis of Caveat Comments

A third basis for choosing between alternative debugging strategies is advice from the annotator and plan-finder on likely errors. The annotator alerts the debugger to oddities in program structure which may be the underlying cause of some semantic violation (section 6.6). The plan-finder fulfills the same purpose with respect to code that contradicts expectations arising from the abstract form of the plan. The mechanism of informing the debugger of the possibly erroneous code is through <u>caveat</u> comments. The comments are noticed when the debugger considers the associated code in the course of debugging some model violation. A repair edit is accorded extra plausibility by the debugger if the correction eliminates the complaint that initiated the caveat.

The annotator generates caveats upon noticing violations of rational form. These are simply sequences of calls to the same primitive such as FORWARD, RIGHT or PENUP. The code is odd: why didn't the user simply coalesce them into a single call with a larger input or, in the case of PENUP, include only the first instruction? The answer may be that the user has forgotten to insert additional instructions -- for example RIGHT commands -- into a FORWARD sequence. A caveat stating that code may be missing is placed between each pair of elements in the sequence of FORWARD's. A violation of rational form occurs in the following triangle procedure because the user has forgotten the first rotation.

```
TO TRI
10 FORWARD 100 <- (caveat annotator rational-form-violation
                        (sequential statements 10 30))
30 FORWARD 100
40 RIGHT 120
50 FORWARD 100
END
```

An edit that inserts a rotation into such a sequence of FORWARD instructions would eliminate the rational form violation and therefore be preferred in competition with other corrections which do not explain the annotator's complaint. If the debugger corrects the program by eliminating the annotation caveat, then the underlying cause of the error is considered to be "Missing Code".

Caveats generated by the plan-finder are created by noting insertions which are not transparent, global plans which depend on specific runtime environments and linear plans in which main-steps use the same resource such as an assumption about a particular state variable. In an extended system, caveats would be useful for noticing such oddities as round-structured programs which fail to halt and shared free variables.

A non-transparent insert is an example of a plan-finding caveat: it may be intended but it is probably a bug. Consequently, FINDPLAN generates a caveat associated with the code following the insert. The caveat declares that if a correction is made that has the effect of making the insert transparent, then the correction should be moved into the final cleanup of the code for the insert. An example occurs in correcting the crooked body of NAPOLEON. The plan-finder produces the following comment for the interface code following the ARMS:

```
TO NAPOLEON
10 VEE
20 FORWARD 100
30 VEE      <- (caveat plan-finder (not (rotation-transparent vee)))
40 FORWARD 100
50 LEFT 90
60 TRICORN
END
```

When the debugger proposes to add a rotation as line 35 which has the effect of making VEE transparent, the caveat is noticed. The debugger is then in a position to accomplish the required state edit and eliminate the cause of the caveat by inserting the edit into VEE as its last line. Thus, the caveat has served to change a linear debugging strategy of editing the interface between the arms and the body into a non-linear edit directly into VEE.

Comments are used -- rather than the Annotator or Plan-Finder immediately calling the Debugger to correct the violation -- because a violation of rational form is not a guarantee of a bug: the oddity may be harmless or even intended by the programmer. An example in which a sequence of FORWARD instructions arises naturally is the following triangle program:

```
TO TRI
10 FORWARD 50
20 FORWARD 50
30 RIGHT 120
40 FORWARD 100
50 RIGHT 120
60 FORWARD 100
END
```

The first two FORWARD's are surprising. However, if this TRI is being debugged in preparation for being converted into a tree program with the trunk inserted between statements 10 and 20 (similar to TREE3, section 2.7), then the apparent violation of rational form is explained. Similarly, an absence of control in a recursive procedure is acceptable if the turtle's halting is not desired. Beginning recursive programs for triangles often take the following form:

```
TO TRI
10 FORWARD 100
20 RIGHT 120
30 TRI
END
```

The utility of comments is that if the code is not suspected of being in error by the debugger, the comment has no effect. The comment has an effect only if the debugging analysis finds a model violation that can be corrected by changing the odd code. It is then that the comment enters the analysis by supporting such a hypothesis with its own complaint about the code. Its complaint -- non-transparent insert or sequential primitives -- can then be used not only to support the plausibility of this debugging strategy but also to suggest the proper repair -- make code transparent or insert interface.

3.8.4 Guessing the Culpable Interface

Even with the restriction to linear edits, fixing a predicate relating two main-steps may produce many possible edits. For example, making the HEAD above the LEGS in NAPOLEON could be done by adding a rotation in any of several places in the program preceding the execution of the TRICORN sub-procedure. To limit the search for the proper edit, the system initially considers edits to only two interfaces -- the interface immediately preceding the second main-step (i.e. code for the model part accomplished last) and the initial setup to the program. The immediate interface is preferred on the expectation that preceding interfaces have already been protected in the course of debugging. The global setup is considered because "Unexpected Runtime Environment" is a common cause of errors. The plausibility of these editing points is then analyzed by the criteria described in the preceding sections -- beneficial side effects, minimal change, and caveats as well as the protection criteria to be described in section 3.10. If they are found implausible, additional interfaces are considered in order proceeding backwards from the second main-step.

3.9 STATE EDITOR

Model-driven debugging ultimately produces a repair strategy consisting of a series of calls to the state editor. Arriving at such a strategy can be complex. Once discovered, however, the state editor then proceeds to make the actual changes to the user's code. The importance of the state editor is that it raises the conceptual level with which the debugger interacts with the actual program. The state editor understands instructions at a higher level than the ordinary LISP and LOGO editors.

Utilizing the imperative semantics of the turtle primitives, it is able to modify code so as to achieve a given picture property. The editor comments its corrections by indicating any assumptions about entry state. Rational form criteria are used to clean up the edit and merge it, where appropriate, with adjacent code.

3.9.1 What The State Editor Does

The purpose of the state editor is to achieve a desired state description or picture property. Examples are:

```
(STATE.EDIT (AFTER (TREE LINE 10)) (= HEADING 90))
(STATE.EDIT (DURING TREE) (= (LENGTH TRUNK) 100))
```

The STATE.EDIT instruction has the following form:

```
(STATE.EDIT <where> <what>).
```

<Where> specifies the location where the edit is to be made.

```
<where> = ((<before, during, after>
           <line, code-group, sub-procedure>)
```

"During" is a request that the edit be true during the code but not effect the exit state: the edit is to be achieved with transparency. An example is having the pen down during a main step, without affecting subsequent main-steps.

The <what> is a state specification in the form of an equality constraining some turtle or picture property to have a given value:

```
turtle-state:  (= <:HEADING, :POSITION, :PEN> <value>)
picture-state: (= <property of picture primitive> <value>).
```

Rotations, vectors and points are picture primitives. They are described during annotation and represent the effects of executing primitive turtle commands. Examples of their properties are endpoints, length and direction for vectors and rays, vertex and degrees for rotations.

3.9.2 How The State Editor Accomplishes Its Goals

Achieving state edits is accomplished in two steps. The first inserts into the code at the specified location the necessary turtle primitives to cause the state to assume the desired value. The second step cleans up the edit by applying Rational Form criteria.

> An interesting fact is that the system produces state edits by generating a plan and then debugging it. The system thus exhibits the same style of planning-debugging in its own thinking as it does in analyzing a user's code.

Each turtle primitive has semantics associated with it that describe its effect. This is the same core of knowledge used to generate annotations. The state editor, however, uses this knowledge in an inverse fashion from the annotator. Rather than asking what a given turtle instruction produces; it inquires what turtle instruction can be used to achieve a specific effect.

```
(PENUP) => (:PENSTATE <- :UP)

(RIGHT :A) => (:HEADING <- (+ :HEADING :A))
(RIGHT :A) => (= (DEGREES ROTATION) :A)

(FORWARD :R) => (:POSITION <- (NEWPOSITION :R :HEADING))
(FORWARD :R) => (= (LENGTH VECTOR) :R)
```

See chapter 6 on Annotation for more detail.

The state editor is generally asked to achieve a given absolute state at a desired location. Turtle commands, however, effect the world relative to their entry state. Hence, in addition to knowing which primitive alters which state variable, the system must know the current entry state. This information is obtained from the annotator. The input to the appropriate primitive is then simply

(DIFFERENCE <desired state> <old state>).

The assumed entry state is protected by an "Assumption" comment.

Upon inserting a line of code, the state editor invokes Rational Form Criteria to criticize the relation of the insert to adjacent code. The rational form criteria are local and they do not criticize the overall organization of the program. Hence, they can remain ignorant of the model. They observe local oddities.

RATIONAL FORM CRITERIA

violation:	Sequences of adjacent forwards
fix:	Delete all but the first translation. Edit the input of the first translation to be the sum of the inputs of the sequence.
violation:	Sequence of rotations separated by non-movement commands.
fix:	Delete all but the first rotation. Edit the input of the first rotation to be the sum of the inputs of the sequence.
violation:	Sequence of similar pen commands.
fix:	Delete all but the first command of the sequence.

Mergers are not forced if adjacent similar commands are described by different purposes. For example, two adjacent FORWARD instructions are not merged if they draw vectors which are pieces of different model parts.

> MYCROFT illustrates in many ways the multiple use of knowledge. Here we see recurrent use of the semantics for procedural primitives. The criteria of rational form are similarly used in many ways. The editor uses it to debug its plans while the annotator uses it to generate caveats for possible structural programming errors. This provides evidence that the system has isolated fundamental types of knowledge related to understanding and debugging programs.

3.9.3 Extensions To The State Editor

Debugging round-structured programs would require an editor capable of accepting high level instructions to modify the recursive or iterative mechanisms in the user's program. An example might be the request to modify a program so that it iterates N rather than N+1 times, for a given initialization of the loop. Such complexity is clearly manageable but, nevertheless, one step above the local modifications made by the turtle state editor.

Achieving a certain state for every round of an iterative or recursive program requires the ability to describe schematically the behavior of the loop. The state upon entry to the round changes with each invocation. The system must understand the way in which one round affects the next. It may also have to take special precautions to properly handle the first or last instances. The technique of achieving the proper edit could be either to solve for the first instance and hope the remaining invocations are also satisfied; or, preferably, to understand the entry condition to a generic round abstractly and be able to make the fix for all rounds with confidence.

For programs with inputs, it is possible that the state comparison occurs at the level of the schematic commentary. (See section 6.5 on Schematic Description.) The schematic commentary describes formal performance, independent of input binding. If the request is itself abstract, then this is obviously necessary. For example, demanding that the length of a vector be equal to a previous vector can be accomplished by identical formal inputs. If the state comparison occurs at the level of the process, comments for protecting assumptions about input bindings should be generated.

3.10 ASSUMPTION AND PROTECTION

DEBUG generates assumption and protection commentary associated with each repair to aid in resolving difficulties when an edit causes new violations or undoes the effects of some previous edit. Assumptions about the entry state at the repair-point describe expectations on which the imperative semantics based their analysis. Protection commentary guards the code from the repair-point to the manifestation-point (the place in the code at which the sub-pictures referred to by the violated model predicate were completed), again because the details of the repair depend upon the state manipulations of the code between the edit and the manifestation-point. Protection is introduced by Sussman in the context of debugging blocks world programs [Sussman 1973].

A simple example arises for the following tree program:

```
TO TREE4         <- (accomplish tree)
10 TRIANGLE      <- (accomplish top)
20 RIGHT 60      <- (setup heading such-that
                      (overlap (interface statement 30) (side 3 top)))
30 FORWARD 50    <- (retrace (side 3 top))
40 RIGHT 45      <- (setup heading for trunk)
50 FORWARD 100   <- (accomplish trunk)
END

TO TRIANGLE      <- (accomplish equitri)
10 FORWARD 100   <- (accomplish (side 1 triangle))
20 RIGHT 120     <- (accomplish (rotation 1 triangle))
30 FORWARD 100   <- (accomplish (side 2 triangle))
40 RIGHT 120     <- (accomplish (rotation 2 triangle))
50 FORWARD 100   <- (accomplish (side 3 triangle))
                    (cleanup position)
60 RIGHT 120     <- (accomplish (rotation 3 triangle))
                    (cleanup heading)
END
```

See figure 3.12 for the picture drawn by TREE4 with the turtle starting at the center of the screen and with a heading of zero degrees.

Debugging the base of the TOP to be horizontal results in the

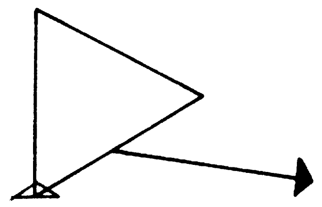

TREE4
VERSION 1
Slanted base & trunk

FIGURE 3.12

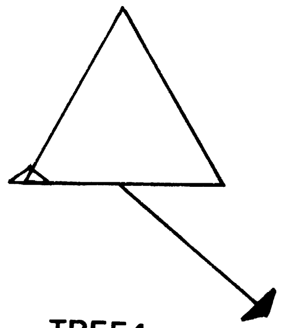

TREE4
VERSION 2
Base made horizontal

FIGURE 3.13

addition of statement 5 to TRIANGLE which rotates the triangle so that the necessary orientation is established.

```
  5 RIGHT 30 <- (setup heading such-that (horizontal (side 3 top)))
```

This produces figure 3.13. Debugging the TRUNK to be vertical by modifying the initial setup, however, undoes this correction (figure 3.14).

```
  3 RIGHT 45 <- (setup heading such-that (vertical trunk))
```

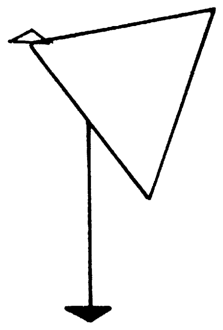

TREE4
VERSION 3
Trunk made vertical

FIGURE 3.14

The solution is for the initial correction of (HORIZONTAL (SIDE 3 TOP)) to include commentary explaining its purpose, scope and assumptions.

Specifically, this commentary is:

1. An assumption that the entry state to statement 5 is :HEADING=0:
   (ASSUME (TREE4 STATEMENT 5) (= :HEADING 0)).

2. A protection to any modifications of :HEADING from statement 5, the repair-point, to statement 50 of TRIANGLE, the manifestation-point of the error:
   (PROTECT :HEADING UNTIL (TRIANGLE STATEMENT 50)).
   Statement 50 is the manifestation-point of the error since it accomplishes (side 3) and INTERPRET is then able to recognize that a violation exists, i.e. that the base of the triangle is not horizontal.

These comments force the debugger to prefer the alternative repair strategy of making the trunk vertical by editing the rotation of statement 40 to be RIGHT 90.

A second use of this commentary, in addition to preventing conflicts between edits, is to simplify debugging the procedure if it is ever run in a new environment. Unsatisfactory initial state values are immediately noticed by the assumption commentary. For example, if statement 5 of TREE4 contains the assumption that the entry heading should be 0, then being run in any other environment will generate a violation. This violation then directs the debugging.

> Thus, previous debugging sessions produce commentary whose specificity eliminates complex questions of responsibility and interpretion. The system has, in effect, generated the snapshots of performance which Naur and Floyd utilize to verify programs [Floyd 1967, Naur 1967].

The assumption comment is passed to the debugger as an instruction and the result is that code is added prior to statement 5 which converts the heading to the desired value.

Often a protection conflict can be resolved. The debugger is simply recalled to achieve the edit which gave rise to the protection, taking into consideration the new entry or exit state requirements. This second call to the debugger involves less effort than the first. The commentary from the first remains and indicates the desired Cartesian state to be achieved at the manifestation-point. If the second edit succeeds without causing unfixable violations as side effects, then the system has patched its own edit and need not reject the basic form of its current analysis.

3.11 SUMMARY OF DEBUGGING CONCEPTS

The debugger's knowledge divides into two categories: general debugging technique and specific imperative knowledge of logic and geometry.

Debugging Technique

1. Linear Attack -- First verify main-steps privately. Then analyze relations in terms of interfaces. Only if all else fails, modify main-steps to fix relations.

2. Plausible Search -- Compare alternative debugging strategies using plausiblity criteria of minimal change to the user's code and maximal beneficial side effects.

3. Culpable Interfaces -- Prefer either the initial interface or the interface immediately preceding the bugged module. This is based on the assumption that the temporal attack has already verified intermediate interfaces.

4. Caveats -- Use caveat comments generated by the Plan-Finder and Annotator to suggest the location of the repair.

5. Intrinsic versus Extrinsic Errors -- Classify model violations as intrinsic or extrinsic on the basis of whether the error is internal to the code being examined. Intrinsic errors have limited scope and can be debugged privately.

6. Handling Multiple Bugs -- Debug those violations of most-limited scope first: that is, debug properties before relations; then intrinsic predicates before extrinsic ones, and finally in temporal order.

7. Commentary -- Use commentary to express the purpose, assumptions and scope (protection) of a correction and to notice conflicts between different corrections.

Knowledge of Geometry and Logic

1. Imperative Semantics of Predicates -- In addition to standard verification code, primitives have semantics that suggest what to do to make the predicate come true. This consists of procedural knowledge which examines code and generates edits to make a particular geometric predicate true.

2. Rigid Body Theorem -- This theorem is a precise statement of the effect of the initial environment on a segment of code for Fixed-Instruction Turtle Programs, namely that the code produces a rigid body and that the initial environment affects only the orientation and position.

3. Imperative Knowledge for Logical Predicates -- Procedures for making conjunction, disjunction, negation, equality and set membership true with minimal effort.

3.12 CLASSIFICATION OF BUGS

The following taxonomy of bugs summarizes the types of errors which the system debugs. The classification is independent of the geometric details of the turtle world and provides general guidance for finding an appropriate fix and repair locus for a violation. The specific details of the state change made to the code is determined by the imperative semantics for the violated model predicate.

Linear Main-Step Failure:
- Manifestation: Failure of main-step to accomplish model part in private, i.e. when run independently.
- Fix: (Private Debugging) Repair in private, rejoin and satisfy any initial assumptions.
- Ex: (NOT (EQUITRI TRICORN)) in NAPOLEON.

Preparation Error:
- Manifestation: Violation of relation between model parts.
- Fix: (Linear Debugging) Find culpable interface, make edit suggested by the imperative semantics for the predicate, and protect assumptions and behavior until the point at which the error was manifest.
- Ex: See Unexpected Runtime Environment and Local Preparation Errors

Unexpected Runtime Environment: (type of preparation failure)
    Manifestation: Violation due to false assumptions of
        the entry state to program. (Program does succeed in
        certain environments).
    Fix: Add an initial setup which converts the actual entry
        state to the desired entry state.
    Ex: (NOT (BELOW LEGS ARMS)) in NAPOLEON.

Local Preparation Error: (type of preparation error)
    Manifestation: Violation intrinsic to the program,
        and not dependent on the initial environment.
    Fix: Modify state appropriate to the imperative semantics
        for the violated predicate.
    Ex: (NOT (VERTICAL TRUNK)) in TREE4.

Non-Linear Main-Step Failure:
    Manifestation: Main-step succeeds in private.
    Fix: See resource conflicts, insertion errors,
        and global errors described below.

Unconsidered Second-Order Constraint on Main-step:
    (type of non-linear main-step failure)
    Manifestation: Violation of a property of a model part
        not detected in private. Manifested by analysis
        of a relation between the main-step and some
        other model part.
    Fix: Modify main-step in such a way that violation is
        corrected while the first-order description of properties
        asserted in the model is still satisfied. Guidance is
        provided by the imperative semantics for the predicate.
        Examples of such transformations are dilation and
        reflection.
    Ex: (NOT (INSIDE MOUTH HEAD)) in BIG.MOUTH.

Resource Conflict: (type of non-linear main-step failure)
    (Mentioned for completeness: not handled by debugger.)
    Manifestation: Violation of a property of a part
        described in the model which was not exhibited in private.
    Fix: Some assumption made when run privately is being
        violated in public. Such an assumption could be the
        availability of a given resource, e.g. a free variable.
    Ex: Attempt to correct both (VERTICAL BODY) and
        (HORIZONTAL (SIDE TOP)) in TREE4 by modifying the
        initial interface statement 5 (section 4.6)

Insertion Error: (type of non-linear main-step failure)
    Manifestation: Main-step failure not indicated in private
        with the additional element that a caveat comment
        generated by the plan-finder informs the debugger
        that the code group for the main-step surrounds an
        insert which is not transparent.
    Fix: Make insert state-transparent.
    Ex: (NOT (LINE BODY)) in NAPOLEON.

Global Error: (type of non-linear main-step failure)

Manifestation: Model part accomplished non-locally fails.

Fix: Find relevant theorem which was the basis of expecting the global plan to succeed. Find assumptions made by theorem which were not justified. Make these assumptions true.

Ex: (NOT (LINE (SIDE 1 TRICORN))) in NAPOLEON.

CHAPTER 4 -- DEBUGGING EXAMPLES

This chapter provides examples to illustrate the utility of the debugging concepts described earlier. This will include debugging scenarios for NAPOLEON, FACE1 (a face with a non-linear bug) and FACEMAN (a stick figure with an incorrectly connected head). If, at any point, the reader feels overwhelmed with details, he should skip to the next chapter which discusses debugging from a broader perspective.

4.1 DEBUGGING NAPOLEON

This section provides a debugging scenario describing MYCROFT's correction of NAPOLEON. Recall that the NAPOLEON procedures with their associated plans are:

```
TO NAPOLEON          <- (accomplish man)
10 VEE               <- (accomplish legs)
20 FORWARD 100       <- (accomplish (piece 1 body))
30 VEE               <- (insert arms body)
40 FORWARD 100       <- (accomplish (piece 2 body))
50 LEFT 90           <- (setup heading for head)
60 TRICORN           <- (accomplish head)
END

TO VEE               <- (accomplish v)
10 RIGHT 45          <- (setup heading for l1)
20 BACK 100          <- (accomplish l1)
30 FORWARD 100       <- (retrace l1)
40 LEFT 90           <- (setup heading for l2)
50 BACK 100          <- (accomplish l2)
60 FORWARD 100       <- (retrace l2)
END

TO TRICORN           <- (accomplish equitri)
10 FORWARD 50        <- (accomplish (piece 1 (side 1)))
20 RIGHT 90          <- (accomplish (rotation 1))
30 FORWARD 100       <- (accomplish (side 2))
40 RIGHT 90          <- (accomplish (rotation 2))
50 FORWARD 100       <- (accomplish (side 3))
60 RIGHT 90          <- (accomplish (rotation 3))
70 FORWARD 50        <- (accomplish (piece 2 (side 1)))
END
```

Figure 4.1 shows the intended stick figure and the bugged pictures produced by these programs.

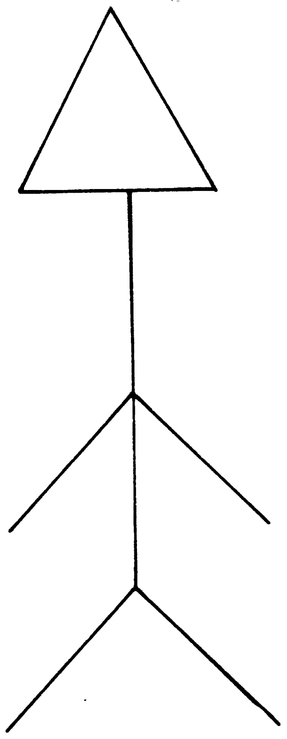
Interpreting the programs with respect to their models (given earlier in section 1.3) produces the following list of violations, shown in ordered form:

```
;Violations of Properties
  ;Intrinsic Property
    (NOT (EQUITRI TRICORN))

  ;Extrinsic Property
    (NOT (LINE BODY))

;Violations of Relations, ordered by execution sequence.
    (NOT (BELOW LEGS ARMS))
    (NOT (BELOW ARMS HEAD))
```

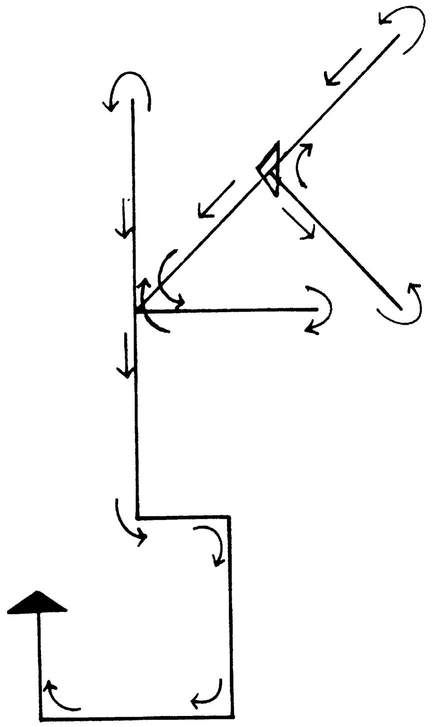
Correcting TRICORN is simplified by the explicit description of the rotations by the EQUITRI model. The debugger has sufficient geometric knowledge of regular polygons such that if the rotations were only described as being equal, then could still deduce that they must be 120 degrees.

Fixing the crooked body is more complex. The difficulty arises because the linear debugging approach of treating main-steps as inviolate is inappropriate. The imperative semantics for LINE direct the debugger to insert a rotation to make the two pieces of the body parallel. The natural place to do this is as statement 35 in NAPOLEON immediately prior to the second piece of the body.

Ordinarily, this is what would occur. However, the debugger notices a caveat comment criticizing statement 30 for not being a rotation-transparent insertion. The result is that the debugger considers adding the rotation into VEE as the last line of code. This eliminates the

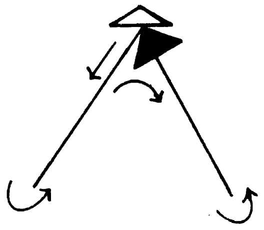


Intended stick figure



Picture drawn by NAPOLEON

Picture drawn by VEE



Picture drawn by TRICORN

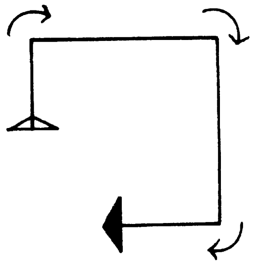
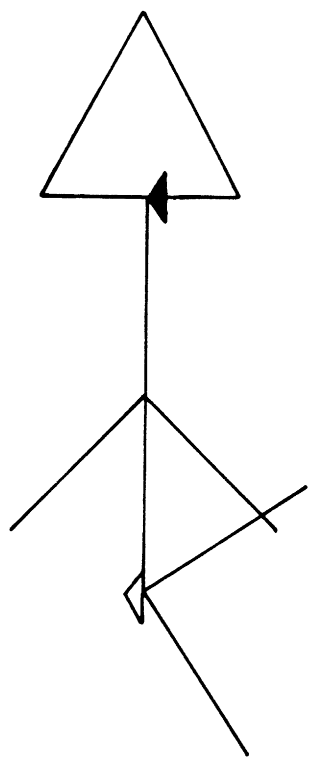


FIGURE 4.1

complaint and makes VEE rotation-transparent. Consequently, this is the preferred debugging strategy.

```
TO VEE          <- (accomplish v)
                   (state-transparent vee)
10 RIGHT 45     <- (setup heading)
20 BACK 100     <- (accomplish l1)
30 FORWARD 100  <- (cleanup position)
40 LEFT 90      <- (setup heading)
50 BACK 100     <- (accomplish l2)
60 FORWARD 100  <- (cleanup position)
70 RIGHT 45     <- (cleanup heading)
END
```

It remains to correct the BELOW relations. Treating main-steps as inviolate and fixing relations by modifying setup steps limits the repair of (BELOW LEGS ARMS) to three possible repair-points: (1) before the legs as statement 5, (2) before the first piece of the body as statement 15 and (3) before accomplishing the arms as statement 25. For BELOW, the imperative semantics direct DEBUG to place the legs below the arms by adding rotations at the setup steps. More drastic modifications to the user's code are possible such as the addition of position setups which alter the topology of the picture; however, MYCROFT tries to be gentle to the turtle program (using the heuristic that the user's code is probably almost correct) and considers these larger changes to the program only if the simpler edits do not succeed. The first setup location considered is the one immediately prior to accomplishing the arms. Adding a rotation as statement 25, however, does not correct the violation and is therefore rejected. The next possible edit point is as statement 15. Here, the addition of RIGHT 135 makes the legs PARTLY-BELOW the arms and produces figure 4.2. This edit is possible but is not preferred both because the legs and arms now overlap and because the legs are not COMPLETELY-BELOW the



NAPOLEON with Line 15 RIGHT 135

FIGURE 4.2

arms. MYCROFT is cautious, being primarily a repairman rather than a designer, and is reluctant to introduce new connections not described in the model. Also, given a choice, MYCROFT prefers the most constrained meaning of the model predicate. If the user had intended figure 4.2, then further model description would be necessary such as (CONNECTED LEGS ARMS) and (PARTLY-BELOW LEGS ARMS).

Adding RIGHT 90 as statement 5 achieves (COMPLETELY-BELOW LEGS ARMS) and produces the intended picture. This correction has beneficial side effects in establishing the proper relationship between the head and arms, confirming for MYCROFT that the edit is reasonable, since a particular underlying cause is often responsible for many bugs. Thus, the result of

(DEBUG (BELOW LEGS ARMS)) is the addition of statement 5 plus the associated commentary to the NAPOLEON program.

```
TO NAPOLEON      <- (accomplish man)
 5 RIGHT 90      <- (setup heading such-that (below legs arms)
                                             (below arms head))
                 <- (assume (= (entry heading) 270))
10 VEE           <- (accomplish legs)
20 FORWARD 100   <- (accomplish (piece 1 body))
30 VEE           <- (insert arms body)
40 FORWARD 100   <- (accomplish (piece 2 body))
50 LEFT 90       <- (setup heading for head)
60 TRICORN       <- (accomplish head)
END
```

The _assume_ comment records the entry state with respect to which the edit was made. If the program is run at a future time in a new environment, then debugging is simplified. The cause of the violation is immediately seen to be an incorrect assumption, and the corresponding repair is obvious. This illustrates the existence of levels of commentary between the model and the program, each layer being more specific, but also more closely tied to the particular code and runtime environment of the program.

4.2 DEBUGGING A BIG MOUTH

This section provides an example of a face procedure which requires non-linear debugging. Figure 4.3 illustrates the intended face. Shown below is the model provided to describe this face.

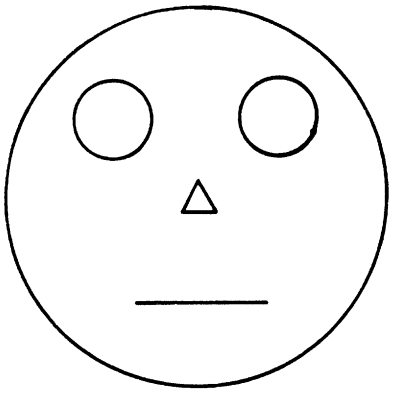


FIGURE 4.3

```
MODEL FACE
M1 PARTS LEFT.EYE RIGHT.EYE NOSE MOUTH HEAD
M2 CIRCLE (HEAD LEFT.EYE RIGHT.EYE)
M3 EQUITRI NOSE
M4 LINE MOUTH
M5 INSIDE (LEFT.EYE RIGHT.EYE NOSE MOUTH) HEAD
M6 ABOVE (LEFT.EYE RIGHT.EYE) NOSE
M7 BELOW MOUTH NOSE
END
```

The following program FACE1 is intended to accomplish this face but actually draws figure 4.4. The main-steps are underlined.

```
TO FACE1               ;version 1
10 SMALLTRIANGLE       <- (accomplish nose)
20 BIGCIRCLE           <- (accomplish head)
30 LEFT 45             <- (setup heading (for interface left.eye))
40 FD.UP 100           <- (setup position for left.eye)
50 SMALLCIRCLE         <- (accomplish left.eye)
100 BK.UP 100          <- (cleanup position such-that (at home))
110 RIGHT 90           <- (setup heading (for interface right.eye))
120 FD.UP 100          <- (setup position for right.eye)
130 SMALLCIRCLE        <- (accomplish right.eye)
140 BK.UP 100          <- (cleanup position such-that (at home))
150 LEFT 45            <- (cleanup heading such-that (at home))
160 RIGHT 180          <- (setup heading for (interface mouth))
170 FD.UP 50           <- (setup position for mouth)
180 RIGHT 90           <- (setup heading for mouth)
190 FORWARD 100        <- (accomplish (piece 1 mouth))
200 BACK 200           <- (accomplish (piece 2 mouth))
END
```

FD.UP and BK.UP in FACE1 are pen-transparent procedures that do a FORWARD and BACK respectively with the pen up. The SMALLTRIANGLE has sides of 20 and is a state-transparent, right-turning procedure. The circle sub-procedures are not fixed-instruction but are written as iterative loops. However, they shall simply be treated as inviolate sub-procedures. Since they are, in fact, correct: debugging of FACE1 will be possible. The circle procedures are state transparent and begin at the center. The SMALLCIRCLE has a radius of 20; and the BIGCIRCLE a radius of 100.

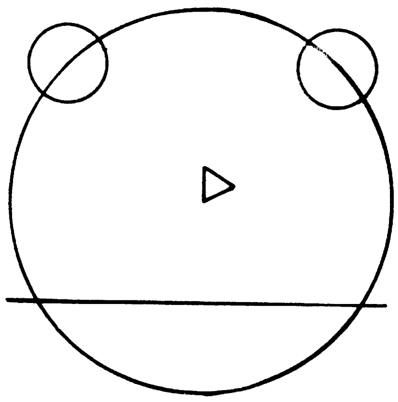
In FACE1, the cleanups are described as returning to a <u>local</u> <u>home</u> state. The existence of this home state is noticed by the Plan-Finder on the basis of observing (1) global connections occurring at this point and (2) the abstract form of the code. Such recognition is very useful for debugging. It can suggest that the underlying cause of a bug is a failure to return to "home".

The first step in debugging is to compare the program's performance to the model's demands and describe the discrepancies. Three violations occur.

```
(NOT (INSIDE LEFT.EYE HEAD))
(NOT (INSIDE RIGHT.EYE HEAD))
(NOT (INSIDE MOUTH HEAD))
```

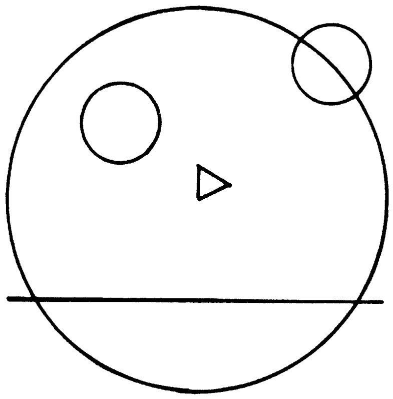
The violations are then debugged. Figures 4.4-4.7 show the stages which the face picture goes through as the code for FACE1 is edited. As we shall see, the violations due to the position of the eyes are linear and are fixed by editing appropriate interfaces. The violation due to the overlap of the mouth and head, however, is not linear and the debugger is forced to edit the main-step for the mouth.

DEBUGGING FACE1



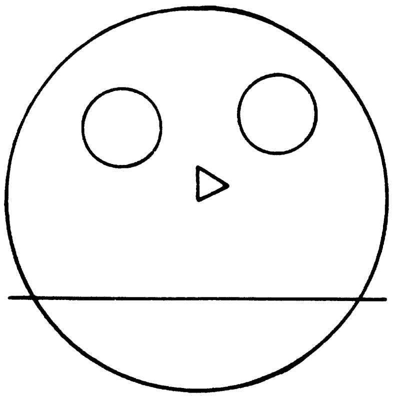
FACE1
VERSION 1

FIGURE 4.4



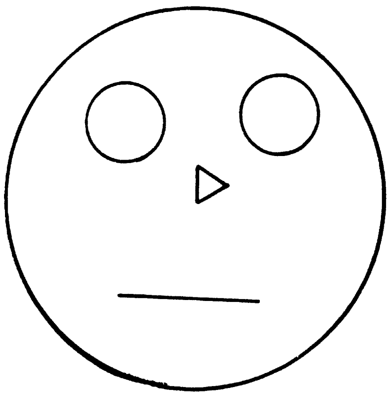
FACE1
VERSION 2
(FIX (INSIDE LEFT. EYE HEAD))

FIGURE 4.5



FACE1
VERSION 3
(FIX(INSIDE RIGHT.EYE HEAD))

FIGURE 4.6



FACE1
VERSION 4

(FIX (INSIDE MOUTH HEAD))

FIGURE 4.7

4.2.1 Fixing the Position of the LEFT.EYE

The earliest violation in terms of the order in which the parts of the FACE are accomplished is (INSIDE LEFT.EYE HEAD). The appropriate part of the imperative semantics for INSIDE is:

```
To make X inside Y, where X is accomplished after Y,
      If 1. the entry position of the interface is inside Y and
         2. the exit position of the interface is outside Y,
      then make the vectors of the interface shorter.
```

The first interface considered for this repair consists of statements 30 and 40, as this is the one immediately preceding the code for the second part, LEFT.EYE. The appropriate change here is to alter the input to the FD.UP instruction of statement 40. The specific input chosen is the average of the maximum and minimum values which satisfy the constraint (rounded to the nearest multiple of 5). In this case, statement 40 can range from FD.UP 30 to FD.UP 80 and hence FD.UP 55 is chosen. Alternatively, the graphic terminal could be used to display to the user the range of permissible values and request his advice. In either case, the commentary recorded is that the particular number chosen is simply the result of selecting from a range of possible values. This allows the number to be easily changed in reaction to subsequent debugging.

```
TO FACE1         ;version 2
         .
         .
40 FD.UP 55      <- (setup position such-that (inside left.eye head))
                    (input chosen between 30, 80)
50 SMALLCIRCLE
100 BK.UP 55     <- (cleanup position such-that (at home))
         .
         .
```

```
;trace of analysis

    ;general debugging technique

        1. Begin with linear debugging, i.e. do not modify
           the main-steps (sub-procedures) for parts.

        2. INSIDE is local.  Therefore, restrict consideration
           of culpable interfaces to those in the causal chain
           between the main-steps for the parts involved in the
           relation.

               a. Culpable interfaces restricted to occuring
                  between code for HEAD and LEFT.EYE.

               b. Ignore TRANSPARENT inserts.  They are not
                  in the causal chain.  Hence, ignore SMALLTRIANGLE.

        3. Possible culpable interfaces are statements 30 and 40.

        4. Debug immediate interface, statement 40.

    ;imperative semantics

        1. For a position interface (statement 40) such-that
               a. origin of interface is inside
               b. termination of interface is outside
           shrink interface.

        2. To shrink, find range which satisfies predicate
           and choose average.
               a. Maximum input such-that predicate is true is 80.
               b. Minimum input such-that predicate is true is 30.
               c. Annotate range.
               d. Shrink input to (average max,min) = 55.

                (STATE.EDIT (STATEMENT 40) (= LENGTH 55))

    ;general debugging technique

        1. Debug consequences of edit.  Statement 40 is a setup
           with an associated cleanup.  Fix cleanup so that
           it succeeds given the new input to the setup.

  (STATE.EDIT (CLEANUP (STATEMENT 100))
    (= (LENGTH (STATEMENT 100)) (LENGTH (SETUP (STATEMENT 40)))))

    ;summary

        (CAUSE - LOCAL PREPARATION ERROR)
        (FIX   - SETUP POSITION SUCH-THAT (INSIDE LEFT.EYE HEAD))
```

4.2.2 Fixing the Position of the RIGHT.EYE

The analysis here is identical to that for the LEFT.EYE. An amusing result is that the the face becomes crosseyed. The minimum distance for the RIGHT.EYE is greater because of the position of the NOSE. The model does not contain any advice that the eyes be at the same level, nor does the system have any innate knowledge of the appearance of faces. Consequently, the setup for the RIGHT.EYE becomes "FD.UP 60", rather than "FD.UP 55".

```
TO FACE1        ;version 3
        .
        .
120 FD.UP 60    <- (setup position such-that (inside right.eye head))
130 SMALLCIRCLE
140 BK.UP 60    <- (cleanup position such-that (at home))
        .
        .
```

If the user is dissatisfied, he must supply additional model specifications. The eyes would be made the same height if the following assertion is supplied:

(RIGHT-OF RIGHT.EYE LEFT.EYE)

The range for statement 60 is computed. But now there is a better choice than simply picking the average value. For the duration of this debugging scenario, we assume that this advice has not been given and our final face will remain slightly crosseyed.

4.2.3 Fixing the Overlap of the MOUTH and HEAD

Usually, debugging a violation between model parts is accomplished by editing the interfaces appropriately. However, this technique is unsuccessful if such edits either fail to eliminate the violation or introduce serious new violations. For FACE1, altering the interface prior to the MOUTH fails to eliminate the overlap. The result is that the system is forced to reject its hypotheses of Local Preparation Error and consider a Non-Linear Main-Step failure, i.e. changing the code for one of the model parts to which the predicate refers.

For the non-linear case, the semantics for INSIDE suggest to either shrink the MOUTH or dilate the HEAD. Dilating the HEAD has bad side effects with respect to connection with the BODY; hence, shrinking the MOUTH is preferred. Shrinking preserves shape and consequently tends not to introduce any new violations. Shrinking is defined procedurally as altering the value of all vector instructions in the code segment by a scale facter less than 1. In this case, the maximum size which the mouth can be is 140, the minimum size 0. The average value of 70 is chosen. The plan indicates that both statements 190 and 200 are pieces of the MOUTH. Therefore, the size of both lines is altered to preserve shape.

```
TO FACE1         ;version 4
         .
         .
180 RIGHT 90
190 FORWARD 35  <- (shrink (piece 1 mouth)
                          such-that (inside mouth head))
200 BACK 70     <- (shrink (piece 2 mouth)
                          such-that (inside mouth head))
END
```

```
;trace

    ;general debugging technique

        1. Begin with a linear attack.  Hypothesize as a Local
           Preparation Error.

    ;This proceeds as for the eyes.  But this time, no interface can
    ;be edited to cause the mouth to be inside the head.
```

2. Reject hypothesis. Attempt non-linear attack.
   Hypothesize as Non-Linear Main-Step Failure.

;imperative semantics for INSIDE

1. Shrink the part to be inside. Compute maximum and minimum size and take the average. In this case, the maximum size is approximately 140. The minimum is zero. Hence, the mouth is to be made of length 70.

```
(STATE.EDIT MOUTH (= LENGTH 70))
```

   The mouth is in two pieces. Both are shrunk so that the total visible length is 70.

;summary

```
(CAUSE - UNCONSIDERED EXTERNAL CONSTRAINT ON MAIN-STEP)
(FIX   - SHRINK MAIN STEP)
```

Deciding that the underlying cause of the the overlap of the MOUTH and the HEAD is that the MOUTH is too large requires a decision as to when the "scale" error first occurred. The linear assumption made by the system is that the error is local to the code directly responsible for the MOUTH, statements 190 and 200. However, it is possible that the error includes statement 170, the setup for the MOUTH. User advice is required to direct the system to extend the scale correction to statement 170.

The FACE is now successfully accomplished.

```
TO FACE1          ;version 4
10 SMALLTRIANGLE
20 BIGCIRCLE
30 LEFT 45
40 FD.UP 55       <- (setup position such-that (inside left.eye head))
50 SMALLCIRCLE
100 BK.UP 55      <- (cleanup position such-that (at home))
110 RIGHT 90
120 FD.UP 60      <- (setup position such-that (inside right.eye head))
130 SMALLCIRCLE
140 BK.UP 60      <- (cleanup position such-that (at home))
150 LEFT 45
160 RIGHT 180
170 FD.UP 50
180 RIGHT 90
190 FORWARD 35    <- (shrink (piece 1 mouth)
                             such-that (inside mouth head))
200 BACK 70       <- (shrink (piece 2 mouth)
                             such-that (inside mouth head))
END
```

4.2.4 Orientation of the Nose

The nose never assumes the proper orientation. The reason is that the user's model does not constrain its orientation. If the user added the following statement to the FACE model

(HORIZONTAL (BOTTOM (SIDE NOSE))),

the system would add the appropriate rotation.

In editing FACE1 such that the bottom of the nose becomes horizontal, the following question would arise: If a rotation is inserted before statement 10 to properly orient the triangle, should it be undone following statement 10? Is the rotation local to the triangle or should its effects be felt throughout the program? The answer depends on the model and plan. In this case, a local home state is a central feature of the plan and, therefore, the rotation would be made transparent.

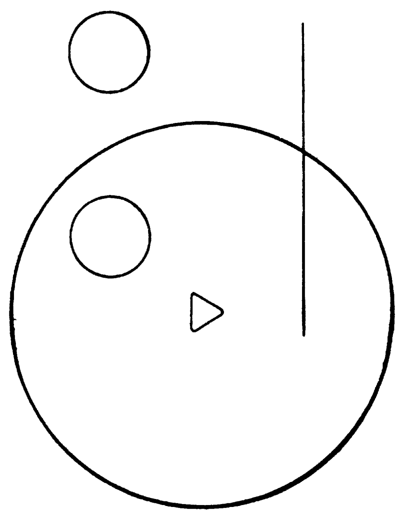
4.2.5 Limitations of Rational Form Criteria

Statements 150 LEFT 45 and 160 RIGHT 180 of FACE1 are sequential primitives of the same type and, therefore, are a violation of Rational Form. However, the rationale for their existence is the use of a "local home" state to which the turtle returns between accomplishing parts. This is observed by the plan-finder. Because it is able to assign distinct purposes to the two code statements, the caveat comment created by the annotator (which was created before plan-finding) is erased.

This code is an example of why the system cannot always consider violations of Rational Form to be bugs. In this case, it is clear to the plan-finder that there was a reason for the apparent violation: namely the goal of returning to the home state before preparing for the next main-step. In other programs, bugs might obscure the existence of the local home and the plan-finder would consequently believe that sequences of the same primitive are unreasonable and generate a caveat comment. This would be an error on the part of the plan-finder; however, the utility of caveat comments is that they do not intrude on the debugging process unless the debugger suspects the cause of a violation to be code in the vicinity of the caveat.

4.3 AN EXAMPLE OF A MODEL-DRIVEN DEBUGGING FAILURE

Suppose the circle procedure used for the head of the face began on the circumference. Let this procedure be CIRCUMCIRCLE. FACE2 utilizes such a procedure and is based on the same plan as FACE1, i.e. a local home at the center of the head is used to interface between main-steps. The underlined statements are differences between FACE2 and FACE1.



NON-TRANSPARENT HEAD
FACE2

FIGURE 4.8

```
TO FACE2
10 SMALLTRIANGLE          ;Draw the nose at the center of the head.
12 FD.UP 100              ;Move to the circumference of the head.
14 LEFT 90                ;Orient turtle tangent to head.
20 CIRCUMCIRCLE           ;Draw the head.  (Left-turning circle.)

   ;The program should now return the turtle to the center
   ;of the head if it is to have a similar plan to FACE1
   ;as intended.  However, it has a bug and the position
   ;cleanup is missing.  The turtle is consequently left on
   ;the circumference of the circle and pointing :WEST rather than
   ;properly returned to the local home.

30 LEFT 45                ;Setup heading for moving to the LEFT.EYE.
        .
        .                 ;Remainder as in FACE1.
```

Following the HEAD in FACE2, the programmer has forgotten to do the required "RIGHT 90" and "BK.UP 100" to return to the center of the face. (This was not necessary in the original FACE1, since BIGCIRCLE was state-transparent with respect to the center of the circle.) The result in FACE2 is that the eyes and mouth are located outside the head. The fixing semantics for OUTSIDE fails to debug this error. Fixing the interface and inserting the "LEFT 90" does not move either the eyes or the mouth inside the head. The occurrence of two errors which must be fixed simultaneously confuses the system.

Planning advice, however, allows the system to succeed. The plan-finder has discovered the user's abstract planning structure of an insertion plan whose skeleton is the center of the head. The fact that following the HEAD, the turtle does not return to this local home causes the plan-finder to generate a caveat comment. When the analysis of the violation of INSIDE leads the debugger to consider the interface (statement 30) following the HEAD as culpable, this comment plays a role. It results in advice to the debugger that the proper correction is to achieve state transparency with respect to the local home described in the plan. The debugger can then go on to correct the INSIDE violation by shrinking the interface.

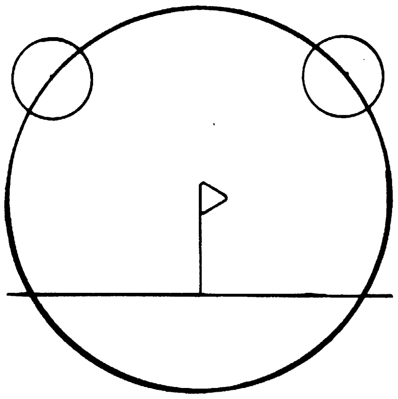
> Cleanup Debugging: If the Debugger suspects the cause of an error to lie where a cleanup was expected by the Plan-Finder (as noted in a caveat), then hypothesize the error to be a "forgotten cleanup" and make a state.edit to insert this cleanup, i.e. a return to the "home". Then continue debugging.

Without the advice of the plan-finder, the debugger could not recognize the necessary intermediate step of returning to the center of the face after drawing the head. Hence, this is an example where purely model-driven debugging is inadequate. The problem with the picture is that it is not able to guide the debugger to the required edits. It is also an example wherein knowledge of plans simplifies the problem and allows successful debugging.

4.4 DEBUGGING A VISIBLE INTERFACE

Let FACE3 be a variant of FACE1 in which statement 170, the setup for the MOUTH, is visible. If the visible code is correctly described as an interface (and not incorrectly assigned to some model part by the plan-finder), then debugging is straight-forward.

```
TO FACE3
        .
        .
160 RIGHT 180          <- (setup heading for (interface mouth))
170 FORWARD 50         ;bug: VECTOR should be invisible.
180 RIGHT 90           <- (setup heading for mouth)
190 FORWARD 100        <- (accomplish (piece 1 mouth))
200 BACK 200           <- (accomplish (piece 2 mouth))
END
```



VISIBLE INTERFACE
FACE3

FIGURE 4.9

INTERPRET would describe the violation as:

(NOT (INVISIBLE (INTERFACE STATEMENT 170))).

Visibility bugs are corrected by retracing and by penstate changes. Recall that the definition of (INVISIBLE V) is:

```
(OR (= (PENSTATE V) :UP) (RETRACE V W)).
```

The more common error is in penstate and that is the correction made here.

```
;trace

    ;debugging technique

        1. Linear debugging.  Bug in property of code.
           The property is local to the code.  Therefore,
           although interfaces are not "main-steps" with
           respect to the picture, the debugging attack here
           is modular and the property can be fixed in private.

              (STATE.EDIT (DURING INTERFACE) (= PENSTATE :UP))

    ;summary

        (CAUSE - PREPARATION STEP FAILURE)
        (FIX   - SETUP PEN SUCH-THAT (INVISIBLE INTERFACE))
```

The result is that statement 170 becomes the required FD.UP.

4.5 FIXING A BROKEN NECK

The following stick figure program FACEMAN illustrates the repair of a connectivity relation, where the connection point is not explicitly described. Suppose FACE1 had been used to draw the head of a stick figure as shown below (assume FACE1 has been debugged):

```
TO FACEMAN              ;version 1, see figure 4.10
10 VEE                  <- (accomplish legs)
20 FORWARD 100          <- (accomplish (piece 1 body))
30 VEE                  <- (insert arms body)
40 FORWARD 100          <- (accomplish (piece 2 body))
50 PENUP                <- (setup pen for interface)
60 FORWARD 50           <- (setup position)
70 PENDOWN              <- (setup penstate for face)
80 FACE1                <- (accomplish face)
END
```

The default meaning for CONNECTED is connection at an endpoint of a

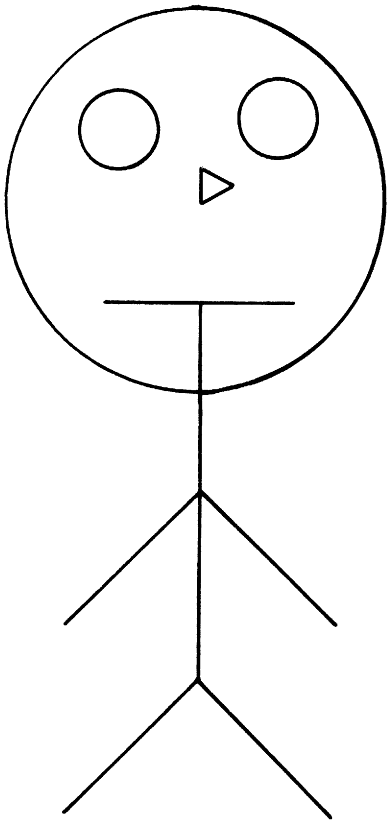


FIGURE 4.10

vector. Hence, it is a violation for the BODY to overlap the HEAD. The imperative semantics direct the debugger to choose the connection point. For the BODY, the desired location is an endpoint. The upper endpoint is the logical candidate as it is the entry state to the interface consisting of statements 50, 60 and 70. This is confirmed by the fact that choosing the lower endpoint of the BODY has bad side effects in terms of the legs and head overlapping. The next step is to choose the point on the HEAD. There is no clear candidate so the system resorts to considering minimal changes to the user's code, in particular, manipulating the input to the interface vector drawn by statement 60. This proves satisfactory. Statement 60 drawing a vector of length 100 causes a connection with the endpoint of the body and there are no bad side effects.

```
TO FACEMAN        ;version 2, see figure 4.11
                  <- (accomplishes man)
10 VEE
20 FORWARD 100
30 VEE
40 FORWARD 100
50 PENUP
60 FORWARD 100  <- (setup position (for FACE1)
                       (such-that (connected body head)))
70 PENDOWN
80 FACE1
END
```

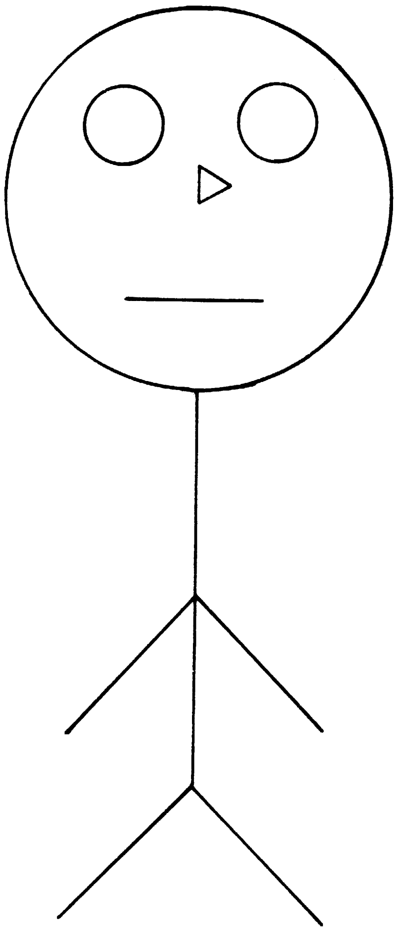


FIGURE 4.11

The edit of altering the position of the HEAD relative to the BODY by editing statement 60, rather than inserting an edit into the FACE1 sub-procedure, reflects a linear analysis wherein the code for sub-procedures is kept inviolate. The Rigid Body Theorem justifies computing the required translation independent of its point of insertion into the program.

```
;trace

    ;debugging technique

        1. CONNECTED is a topological model predicate.
           Therefore, hypothesize as an internal Local Preparation
           Error. The culpable interface is restricted to the
           causal chain between the completion of the BODY
           and the beginning of the HEAD.

        2. For linear debugging of an interface, treat main-steps
           as inviolate. Therefore, the culpable interface is
           further restricted to occuring between BODY and FACE,
           i.e. not inside the FACE1 subprocedure.

    ;imperative semantics for CONNECTED

        1. Compute the required translation. This is done in
           several steps:

                a. Choose a connection point on the second part. Endpoints
                are preferred. In this case, the second part is the BODY.
                The preferred connection point is the upper endpoint.
                Using the lower endpoint results in bad side effects.

                b. Choose a connection point on the HEAD. Since the HEAD
                does not have any endpoints, the strategy is to alter the
                interfaces in a minimal way and observe where the
                connection is produced. The only interface between the
                body and the face is statement 60. The interior of the
                face procedure is inviolate. Decreasing the input to
                statement 60 moves the head down towards the body.
                Connecting the upper endpoint of the body to the
                circumference of the head satisfies the model.

        (STATE.EDIT (BEFORE FACE1)
                (= POSITION (+ POSITION <REQUIRED MOVEMENT>)))

    ;summary

        (BUG - LOCAL PREPARATION ERROR)
        (FIX - SETUP POSITION SUCH-THAT (CONNECTED BODY HEAD))
```

CHAPTER 5 -- OVERVIEW OF DEBUGGING

5.1 HIERARCHICAL DEBUGGING

This chapter considers the debugging problem from a broader perspective. The first observation to be made relates to the relationship between code and commentary. Debugging is correcting a program so as to satisfy its commentary. The problem becomes simpler as the commentary becomes more explicit. Hence, a useful point of view from which to describe the debugging process is to consider competence to debug successively higher levels of commentary.

> The level of the commentary refers to the specificity with respect to the actual code. In this sense, the program itself is the lowest level of commentary.

In the turtle world, the first level of description above the actual code consists of "state descriptions". This commentary can be passed directly as commands to a state editor which appropriately modifies the program. No complex analysis in finding the locus of the correction is necessary. An example is the set of assumptions generated while debugging FACEMAN and TREE1. With this level of expertise assumed, debugging the plan becomes finding the appropriate state commentary. Then, with an understanding of how plans are debugged, the problem of finding the plan given only the model can be considered.

This point of view is hierarchical with each stage assuming the expertise of the preceding one. Notice that this same framework, when viewed from the reverse perspective, can describe a program-writing system: program writing works by successively detailing the level of description until a program is produced.

This hierarchical point of view is only a first approximation. In more complex situations, we should expect two-way communication between the different levels of description. This is the case in the TREE1 example (section 7.7) where the debugger rejects the plan and requests a new one.

5.2 TOP-LEVEL DEBUGGING GUIDANCE

The top-level organization of model-driven debugging is to order the model violations and then proceed to fix them in turn. This technique makes the basic assumption that guidance in fixing the program can be obtained by analyzing the specific details wherein the picture failed to satisfy its description. Alternatively, top-level guidance can be obtained through:

1. structure-driven debugging - insight into the form of programs, e.g. such structural considerations as recursive and iterative control patterns and global versus local variable scope.
2. evolution-driven debugging - the evolutionary or editing history of the user's code.
3. process-driven debugging - the abstract form of the process at the time of the error [Sussman 1973].

A more complete debugging system would exhibit all of these forms of direction.

5.3 STRUCTURE-DRIVEN DEBUGGING

Structure-driven debugging is concerned with "programming errors". These are errors stemming from the use of the language of procedures rather than errors in theory or in planning. This class of errors includes bugs in recursive and iterative control. Examples of such bugs are slip-through errors (in which the end test of a loop fails to catch the counter), fence-post problems (in which the program repeats the wrong number of times), incorrect preparation between rounds, incorrect initialization and wrong-way-counting. The following triangle program is an example of the last type of bug where the counter should be incremented rather than decremented.

```
TO TRI
10 MAKE "SIDES" 0
20 IF :SIDES=3 THEN STOP
30 FORWARD 100
40 RIGHT 120
50 MAKE "SIDES" :SIDES-1   ;Decrements rather than increments counter.
60 GO 20
END
```

This error is noted by a caveat comment which provides later debugging guidance.

Structural errors due to incorrect scoping of variables must be handled when the complexity of programs is extended to allow inputs. Typical errors are variables which are bound that should be free, free that should be bound, and conflicts in names.

Each type of structural error would have a "fixer" that could correct it. For control, an expert on integer arithmetic is useful to decide how many rounds a given program will execute for a particular end test, step function and initialization. (See Ruth [1973] for a discussion of such an expert used to correct errors in simple sorting programs.)

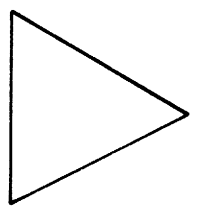
The difficulty in debugging such errors is not in fixing them once known but recognizing that they are the culprits in the first place. Abstract knowledge of recursive and iterative planning is helpful. It is used to find the plan of such programs, of course. It also aids debugging by generating caveats which alert the system to the occurrence of structural errors.

5.4 EVOLUTION-DRIVEN DEBUGGING

Editing errors arise from modifying an existing program to achieve a new purpose. These errors include "incomplete variablization", "violating prerequisites", and "unexpected interactions" (between free variables). The clue to understanding these errors is in the nature of the edits and their purposes.

As an example, a typical bug that occurs when editing a fixed instruction triangle program into a program capable of drawing triangle of arbitrary size is Incomplete Variabilization.

```
TO INCOMPLETELY.SCALED.TRI :SCALE
10 FORWARD 100     ;This line is missing the scale factor.
20 RIGHT 120
30 FORWARD 100*:SCALE
40 RIGHT 120
50 FORWARD 100*:SCALE
END
```



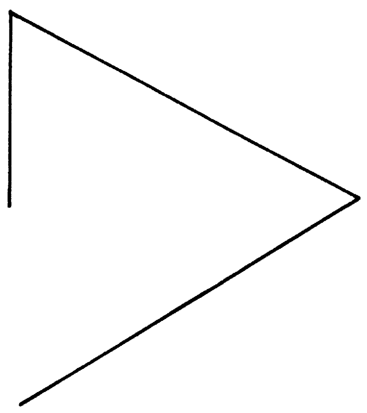


FIGURE 5.1

The evolution of the program can be recorded by noting the purpose of each edit. Obviously, if the program is run in a new environment and fails, the edit is suspected of causing the bug. If the model has not changed, then this hypothesis can be tested by running the earlier version of the program and observing if it succeeds in the new environment.

In finding the plan, multiple possibilites can be generated. Hence, in the first debugging session, the contingency exists that the program is being incorrectly interpreted because the system is using the wrong plan. But, in subsequent sessions, the system can be more and more convinced that it has found the right plan and focus attention on the resulting bugs rather than examining alternative plans.

A mini-world of knowledge that would facilitate this type of debugging is an understanding of the process of learning programming (and the related problem-solving). Typical evolutions of a program are towards greater generality and power. An example is first the addition of a movement input to a SQUARE procedure to accomplish squares of any size and then the insertion of a rotation input thereby generalizing to a POLY program which can draw any regular polygon.

```
TO POLY :SIDE :ANGLE
10 FORWARD :SIDE
20 RIGHT :ANGLE
30 POLY :SIDE :ANGLE
END
```

(See figure B.7 for pictures drawn by this program.) Knowledge of the typical bugs encountered in generalizing, in handling new environments, and in coping with more complex procedural structures would provide insight into the underlying cause of an error.

5.5 PROCESS-DRIVEN DEBUGGING

The idea of process-driven debugging is to observe the current state and history of the process in order to diagnose the cause of a bug. This is one of the basic notions developed in Sussman's HACKER program [Sussman 1973]. The following is an example of this kind of debugging drawn from Sussman's research. Imagine that a one-armed robot has been asked to build the tower of three blocks shown in figure 5.2. This request

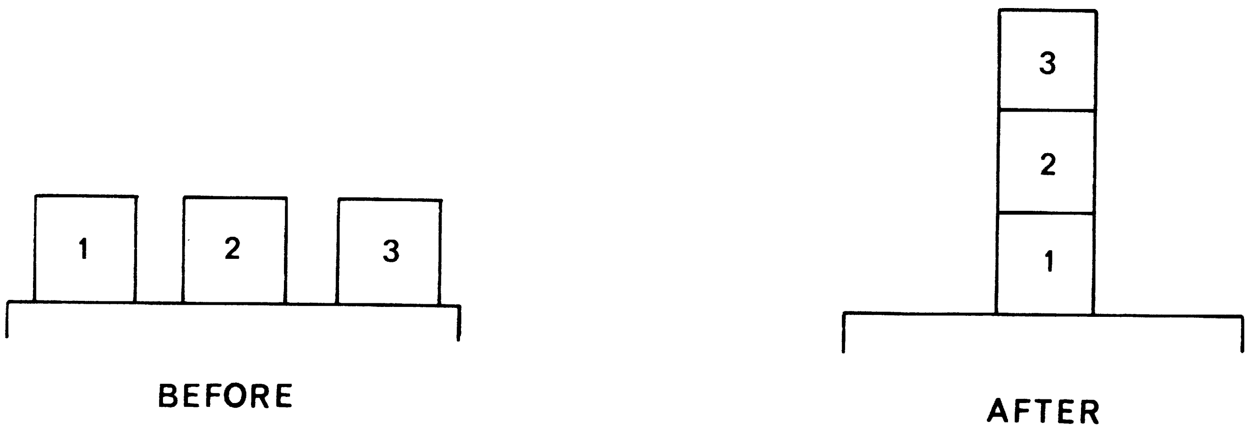


FIGURE 5.2

to the HACKER program is made by asking:

(ACHIEVE (AND (ON A B) (ON B C))).

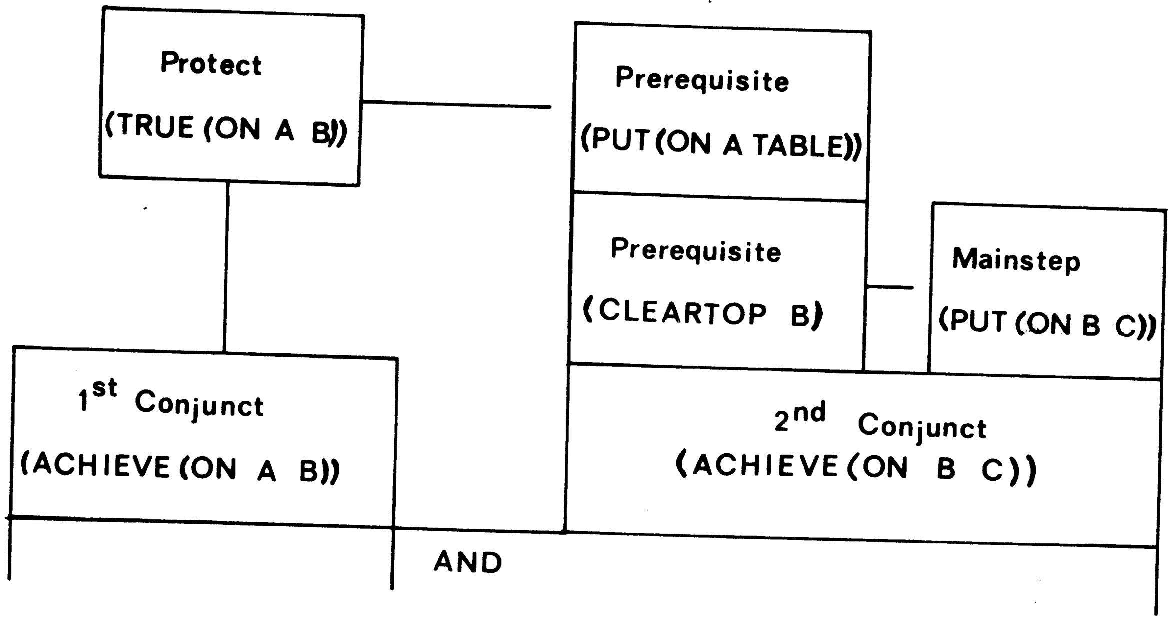
Assuming the robot has had no previous experience with towers, its first approach would be to simply follow the linear plan of:

```
(ACHIEVE (ON A B))
(ACHIEVE (ON B C)).
```

Unfortunately, this plan will fail. Step 1 results with A being placed on B. However, when the time comes to move B onto C, A must be taken off since the robot can only move one block at a time. Thus, in achieving the second main-step, the robot has undone the first.

The manifestation of this disaster is recognized as a violation of a protection created when A is placed on B. The protection asserts that (ON A B) must remain true throughout the remainder of the program. The clearing of the top of B in preparation to moving B onto C conflicts with this protection. The underlying cause of the bug is determined by matching the state of the process against a collection of abstract patterns for different types of bugs.

In this case, the bug pattern is shown in figure 5.3. The lower-



Pattern of PREREQUISITE-CLOBBERS-BROTHER-GOAL

FIGURE 5.3

case labels represent the bug pattern: the upper-case code represents the matching behavior in the program. The cure for such a bug which HACKER applies is to alter the order of the main-steps. The result is that the code for achieving the tower becomes:

```
(ACHIEVE (ON B C))
(ACHIEVE (ON A B)).
```

Note that if we think of the static form of the desired block structure as the model, then the bug is not properly diagnosed. Each of the main-steps in the original program is designed to achieve the placement of a block into the proper relative position for the final goal state. The difficulty is recognizing a constraint not mentioned in the model: namely that support from the bottom up is required. Process-driven debugging is useful for recognizing unexpected interactions between independently designed main-steps.

The turtle bugs which have been analyzed have been mostly of the kind in which code fails to accomplish a predicate asserted in the model. The problems have been ones of missing or incorrectly designed main-steps or preparatory-steps. Bugs in which the code has been properly written but is simply in the wrong order have not been considered. Such bugs are not frequent in the turtle world for a very simple reason: ordering is not as critical as it is in the blocks world. Although ordering may simplify the required preparations, it is always the case that a turtle program can be debugged without reordering the main-steps. This follows from the fact that a main-step can never undo the success of a previous main-step as can happen in the BLOCKS world. In the turtle world, pictures cannot be erased. Therefore, this kind of interaction is impossible.

There are certain kinds of unfavorable interactions in the turtle world. These arise from a main-step (which has been successful in private) failing in the context of the super-procedure because of previous code clobbering certain expected entry-state conditions. The most typical case is when the pen is moved into the :UP state by some previous preparatory-step and not returned to the :DOWN state before execution of a following main-step, resulting in an invisible picture.

The utility of examining the state of the process becomes more important as the programs become more complex. This is because main-step failures due to unexpected interactions become more common as a result of unexpected interactions due to global variables and shared data structures.

5.6 EXTENSIONS WITH A DOMAIN-ORIENTED REASONING PROGRAM

Natural Models are those in which the facts asserted translate readily into local planning statements. These are the type of model given for the trees, wishing wells and stick figures used in the debugging examples. When the model expresses an outlook that is very distant from the plan, (i.e. wherein most model statements are achieved globally), the system is in trouble. There will be great difficulty in finding the plan and understanding the purposes of the program structure. An example would be a model for a face that references all parts to the center of the head coupled to a program that builds the face from bottom to top without the use of a local home state. A reasoning program for processing models would allow greater latitude in model specification. Such global properties as area and symmetry could be understood, i.e. properly related to the program structure.

Examples of some useful geometric facts for understanding and debugging are:

1. (Total Turtle Trip Theorem) A program is heading transparent if and only if the sum of the rotations is 0 (mod 360).

2. (Poly Theorem) The turtle rotations for a regular polygon equal 360/(number of sides).

5.7 THEORY VERSUS PROCEDURE BUGS

Model violations may be due to errors in theory or errors in procedural implementation. This distinction represents two ends of a spectrum and not an absolute, discrete division.

Theory bugs are errors in the understanding of the domain in which the program's effects are to occur. For example, such a bug is assuming an incorrect relation between the length of the inscribed radius and the side of the triangle. They occur in attempting to generate a plan from the model.

Procedure bugs are errors in carrying out a correct plan. This would include local preparation errors, unexpected runtime environments, linear and non-linear main-step failures. This paper is generally concerned with procedure bugs. To correct theory bugs, a theorem-prover is necessary rather than an expert on programming.

Theory bugs blend into procedural bugs. An example is guessing the wrong rotation for causing the turtle to line up with a previously drawn vector. The plan is correct in desiring an overlap. But the understanding of the necessary state relation is in error. The mistake may be theoretical (due to an incorrect understanding of the sum of the rotations in the triangle) or it may be procedural (due to an incorrect understanding of the exit state of the preceding code-segment).

In a more complete model of the program writing and debugging process, both a domain-dependent knowledge system and a programming expert would be present.

5.8 GENERALIZABILITY OF DEBUGGING TECHNIQUES

The mini-world of programs in which MYCROFT operates is that of Fixed-Instruction Turtle procedures. These are, of course, a particularly simple form of program. Their simplicity allows the imperative semantics for the geometric primitives to utilize the Rigid Body Theorem, justifying the same state change to different interfaces to correct a given bug.

The debugging techniques used to handle even these simple programs are by no means exhaustive. Nevertheless, it is worth noting that many of the techniques utilized by MYCROFT are of broad application: an initially linear analysis, the need to order the attack on multiple bugs, competence to cope with alternative debugging strategies--these are useful regardless of the nature of the top-level direction or the complexity of the program.

The debugger has also been designed to extend to more complex geometric programs by describing and repairing bugs in a qualitative way. Picture bugs are not explained in terms of Cartesian coordinates. Such quantitative reasoning is not extendable. Instead, the bug is classified as a preparation or main-step failure. Such abstract reasoning better explains the problem because it tells "why". Indeed, this is the only way to proceed when the model does not fully determine the picture to the extent of actually being able to deduce coordinates.

The choice of plane geometry as the semantic domain for MYCROFT was not accidental. Geometry allows the use of a Cartesian annotator and a powerful model language for specifying spatial relations. Other domains may not be susceptible to a MYCROFT-like approach because of the lack of powerful ways in which to document the effects of the program and the lack of a good model language. However, it is worth noting two points:

1. Spatial models are very important for programming in applications

beyond graphics. (There is [illegible] programmers refer to memory, stacks and data structures in spatial ways.)

and 2. Program planning and debugging involve techniques of broad applicability, [illegible] of domain-dependent knowledge.

CHAPTER 6 -- DESCRIBING PERFORMANCE

6.1 THE DESCRIPTION OF PROGRAMS

Debugging is impossible without good description of a program's purpose and performance. MYCROFT begins with the program and a model describing the intended result. Two forms of additional commentary are then generated: Performance Annotation documents the effect of running the program while the Plan explains the intent. This commentary is organized as sets of assertions in a database, bound together into sequences representing what happened and why. Figure 6.1 shows part of the database generated to describe TREE. The nodes are organized so that the horizontal axis represents time and is used to answer such causal questions as what changes occurred to which state variables and which code was responsible for those changes. Similar data structures for describing programs are used by Fahlman [1973] and Sussman [1973].

The vertical axis represents teleological abstraction and explains the purpose of the code. Models fit into this descriptive framework as the highest level of abstraction. They describe the final goal without ties to specific plans or chronological performance. The next level is the plan, indicating the sub-goal organization for accomplishing the model. Finally, the teleology rests on a description of the actual performance of the turtle program when executed in a particular initial environment.

MYCROFT analyzes a program by first building a complete performance annotation and then applying the plan-finder to assign purposes to the code. Performance annotation is accomplished by running the user's turtle program in a "careful mode" which produces four kinds of description.

1. Process Annotation is a description of the output of the

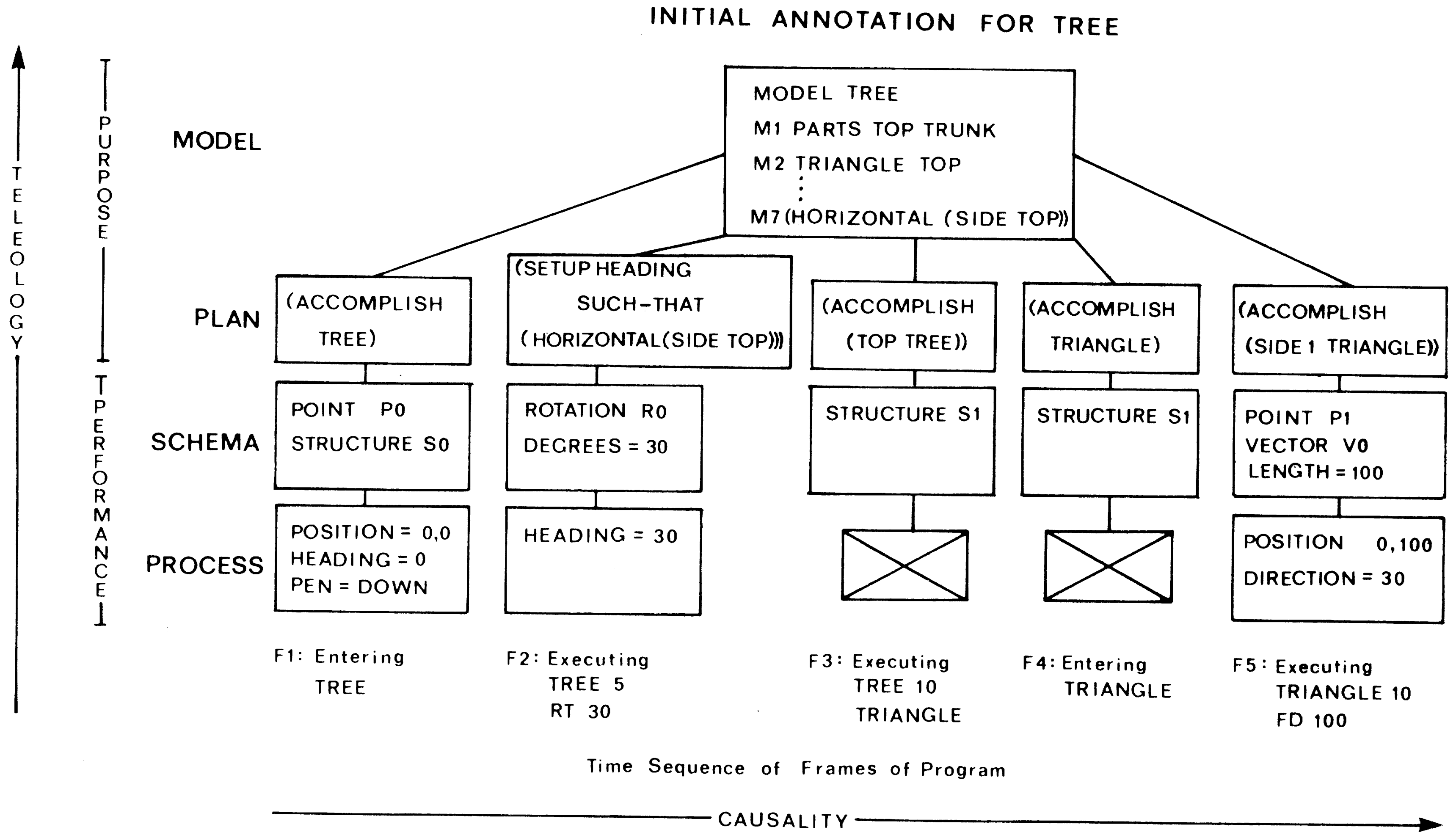

INITIAL ANNOTATION FOR TREE
TELEOLOGY
PURPOSE
PERFORMANCE
MODEL
PLAN
SCHEMA
PROCESS
MODEL TREE
M1 PARTS TOP TRUNK
M2 TRIANGLE TOP
M7 (HORIZONTAL (SIDE TOP))
(ACCOMPLISH TREE)
(SETUP HEADING SUCH-THAT (HORIZONTAL(SIDE TOP)))
(ACCOMPLISH (TOP TREE))
(ACCOMPLISH TRIANGLE)
(ACCOMPLISH (SIDE 1 TRIANGLE))
POINT P0
STRUCTURE S0
ROTATION R0
DEGREES = 30
STRUCTURE S1
STRUCTURE S1
POINT P1
VECTOR V0
LENGTH = 100
POSITION = 0,0
HEADING = 0
PEN = DOWN
HEADING = 30
POSITION 0,100
DIRECTION = 30
F1: Entering TREE
F2: Executing TREE 5 RT 30
F3: Executing TREE 10 TRIANGLE
F4: Entering TRIANGLE
F5: Executing TRIANGLE 10 FD 100
Time Sequence of Frames of Program
CAUSALITY

FIGURE 6.1

program. It consists of a record of the effects of executing each program statement. For turtles, this consists of the creation of vectors, vector structures, rotations and points.

2. Schematic Annotation isolates those properties of the program and its performance which are independent of the initial environment. This is important when debugging requires knowledge of whether an error is due to the initial environment.

3. Planning Advice suggests the segmentation of the program with respect to accomplishing the model on the basis of such criteria as global connections.

4. Debugging Advice describes suspicious code by caveat comments which aid in subsequent debugging.

Details of these four kinds of commentary are given below.

6.2 PROCESS ANNOTATION

Process annotation provides a description of the output of a program and its sub-procedures in terms of some language appropriate to the purpose for which the program was designed. For example, the performance annotation for an arithmetic program might be in terms of mathematical equations to be satisfied at various points in the computation [Floyd 1967]. For turtle programs, an obvious choice is to produce a Cartesian description of the picture drawn by the program. Annotation should reveal the basic effects of the code, free of vagaries of individual programming style. This would include knowing the description of a vector, regardless of whether the actual command is FORWARD, BACK or SETXY. (The last command moves the turtle to an absolute position on the screen.)

Annotation produces a sequence of frames. A frame is generated to describe the execution of each primitive and sub-procedure call. Each frame is a set of assertions specifying (1) any changes to the turtle's state and (2) the properties of any picture elements which have been created. The turtle's state consists of the values of the global variables

:HEADING, :POSITION and :PEN. Picture elements (created as side effects of executing turtle commands) are vectors, rotations, points and structures (vector sets drawn by recognizable code segments such as sub-procedures).

Figure 6.1 represents the annotation for the first five frames of the following TREE and TRIANGLE programs:

```
TO TREE              TO TRIANGLE
5 RIGHT 30           10 FORWARD 100
10 TRIANGLE          20 RIGHT 120
20 RIGHT 60          30 FORWARD 100
30 FORWARD 50        40 RIGHT 120
40 RIGHT 90          50 FORWARD 100
50 FORWARD 100       60 RIGHT 120
END                  END
```

The commentary is generated by a request to ANNOTATE TREE starting at HOME, where HOME is defined by:

```
:POSITION = (0, 0), the center of the screen
:HEADING  = 0 degrees, pointing upwards
:PEN      = "DOWN", the turtle will leave a track.
```

For more complex programs, the initial environment would consist of the input bindings and the values of all free variables.

There are a total of 16 frames in the annotation for TREE as shown in figure 6.2. This basic chronological sequence of frames is called the CHRONTEXT [Sussman 1973] . The value of a state variable in a given frame can be determined by looking back up the CHRONTEXT for the most recent frame in which it was set. This frame will also indicate the code responsible for the assignment. Frames can be chained in different sequences other than the chronological order of execution. For example, F1, F2, F3, F12, ... , F16 describes the performance of TREE, treating the call to the TRIANGLE sub-procedure as a primitive with no internal details. On the other hand, the sequence F4, F5, ... , F11 annotates the performance

FRAME SEQUENCE FOR TREE

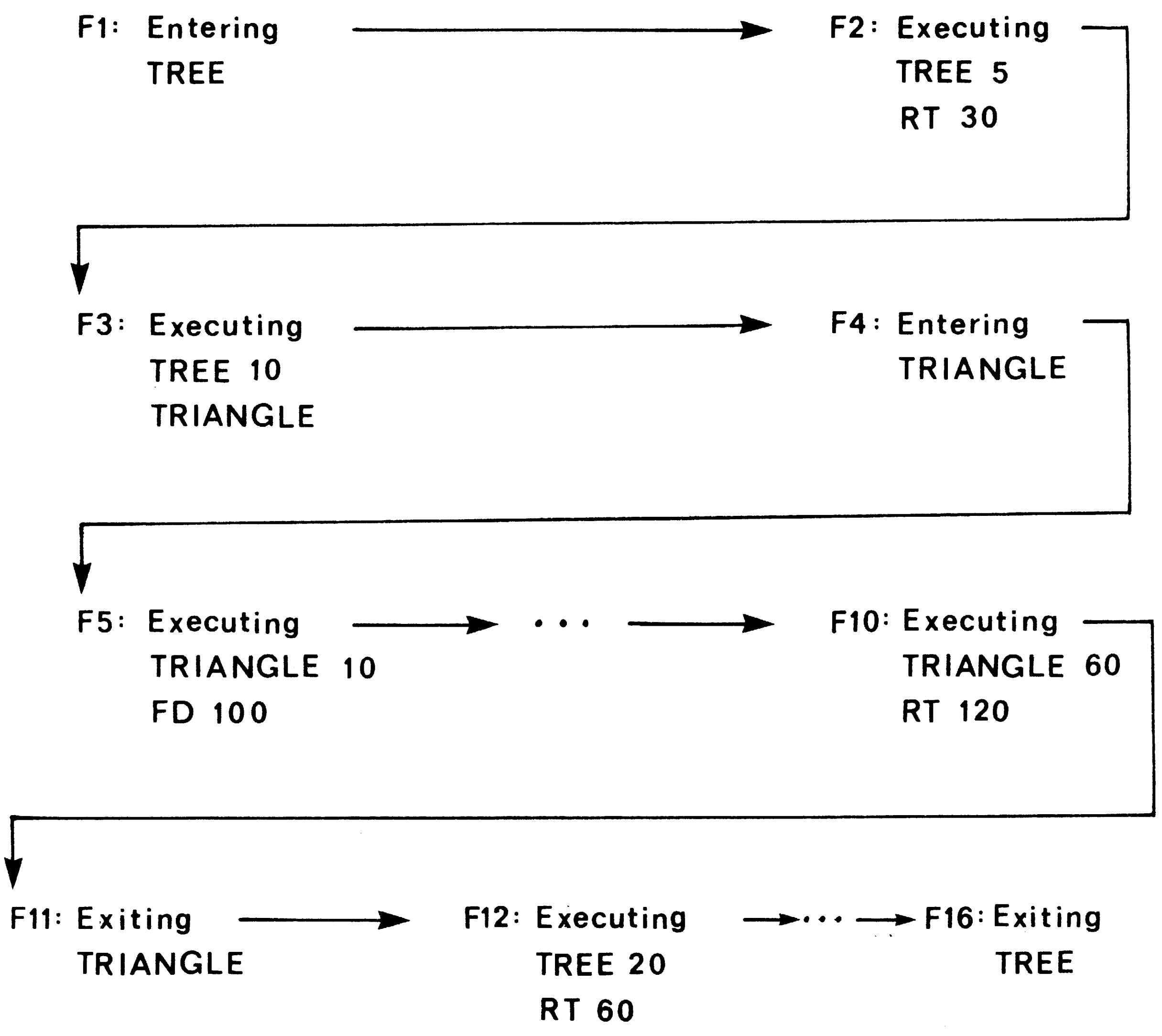


FIGURE 6.2

of TRIANGLE independently (except for entry state) of the remainder of TREE.

6.3 SEMANTICS FOR TURTLE PRIMITIVES

The process annotation is generated by imperative semantics associated with each turtle primitive. These semantics describe the performance of the turtle command.

```
SEMANTICS FOR (FORWARD :DISTANCE)       ;Draws a vector.

        (:VECTOR <-- (GENERATE-NAME 'V))
            ;All vertices, rotations, vectors and structures
            ;are given unique names to facilitate later debugging.
            ;If subsequent investigation reveals that the
            ;particular object has been given a label by
            ;the user, then the system name is replaced by the
            ;user's identifier.

    ;Describe the Vector in terms of its direction and length.

        (ASSERT (= (DIRECTION :VECTOR) :HEADING))
        (ASSERT (= (LENGTH :VECTOR) :DISTANCE))
        (ASSERT (= (VISIBILITY :VECTOR) <PENUP, PENDOWN, RETRACE>)

    ;Update the State of the Turtle

        (:POSITION <-- (FORWARD :DISTANCE))
                ;FORWARD :DISTANCE outputs coordinates of the new
                ;position.  The turtle state variable :POSITION is set
                ;to this new location of the turtle.

        (:POINT <-- (GENERATE-NAME 'P))
                ;If the coordinates are unique, bind :POINT to
                ;a new name for this position.  If not, use the
                ;old name for the position.  If a name already
                ;exists for this position, record the connections
                ;occurring at this point between :VECTOR and
                ;previous vectors.
```

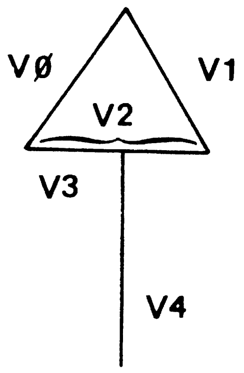
```
SEMANTICS FOR (RIGHT :ANGLE)        ;Rotates the turtle.

        (:ROTATION <-- (GENERATE-NAME 'R))

    ;Describe the Rotation in terms of its vertex and degrees.

        (ASSERT (= (DEGREES :ROTATION) :ANGLE)
        (ASSERT (= (VERTEX :ROTATION) :POSITION)

    ;Update the State of the Turtle

        (:HEADING <-- (RIGHT :ANGLE))    ;RIGHT outputs the new heading.
```

At the level of the process, actual numerical values are determined for the above properties. Because these assertions depend upon the particular state of the initial environment, this is the most specific, least abstract level of commentary when compared with the model, plan and schematic annotation.

6.4 ANTECEDENT COMPUTATION OF MODEL PREDICATES

Certain geometric relations are particularly easy to observe during annotation and are therefore precomputed at that time. Connectivity at endpoints is an example of an important predicate which is simple to compute during annotation. The process description, with its access to numerical coordinates, can observe the turtle returning to a position which was visited earlier. The antecedent computation of connectivity for TREE is illustrated in figure 6.3.

ANTECEDENT COMPUTATION OF CONNECTED



Local Connections

(CONNECTED VØ V1)

(CONNECTED V1 V2)

Global Connections

(CONNECTED V2 VØ)

(CONNECTED V2 V4)

FIGURE 6.3

ANTECEDENT SEMANTICS FOR CONNECTED

1. Record local connections due to the sequential behavior of the turtle.

2. If the turtle returns to a previously visited vertex, then record the connections between the vector just drawn and any previous vectors with endpoints at this vertex. Include the comment that these connections are global.

Similarly, VERTICAL, HORIZONTAL and PERPENDICULAR are precomputed during annotation.

Interior connections where the turtle recrosses some previous vector or ABOVE relations are not noticed during annotation as they depend on global properties of the picture and would be costly to compute. Instead, they are determined only upon request during subsequent interpretation and debugging.

The geometric model predicates are implemented as procedures which expect to find their answer in the database. If it is present as a result of annotation or previous computation, then it is immediately returned. If not, the answer is computed and placed in the database. This is done using FETCHs and IF-NEEDED procedures in CONNIVER [McDermott 1972], although it could equally well be written using PLANNER's THGOAL and CONSEQUENT theorems [Hewitt 1972, Sussman 1971].

6.5 SCHEMATIC DESCRIPTION

The performance annotation can be divided into Process and Schematic description. Process annotation describes the performance of the program in the particular runtime environment in which it was executed. Schematic description treats the environment formally and examines the definition. Properties of the code that are independent of the initial runtime environment are asserted.

Fixed instruction turtle programs generate rigid bodies. The initial environment affects the picture only with respect to its visibility, origin and orientation. Therefore, schematic description is particularly simple. All local properties such as connectivity, inside/outside/overlap, and shape are schematic while properties describing the numerical heading or position of parts are part of the process description. The process annotations of the same program run in two different environments are related by a change of coordinates consisting of a translation and rotation.

For the turtle semantics of section 6.3, the creation of vectors and descriptions of their length are schematic properties while the coordinates of their endpoints and their direction are part of the process annotation. Refer again to figure 6.1 for an example of this distinction.

The importance of the schematic description for debugging is that the system is assured that the runtime environment is irrelevant to schematic properties. The debugger should not consider altering the initial setup to fix a bug in a schematic property such as connectivity. On the other hand, fixing the initial setup for an unexpected runtime environment becomes a common debugging method when the property is process dependent as is the case, for example, for HORIZONTAL and VERTICAL.

The need for schematic descriptions becomes much more important when the programs are allowed inputs, iteration and recursion. Two techniques for generating schematic descriptions of more complex programs -- Structural Analysis and Formal Execution -- are discussed in section 6.8.

6.6 PLAN-FINDING ADVICE

Although performance annotation does not examine the model, it can reveal clues to the grouping of the user's program into main- and preparatory-steps which aid in finding the plan.

1. Sub-procedures that draw visible sub-pictures are hypothesized to be main-steps that accomplish some model part.

2. Maximal sequences of "invisible" primitives such as (a) vectors drawn either by retracing or with the pen up, (b) rotations, and (c) PENUP commands are grouped together as possible preparatory-steps.

3. Maximal sequences of visible vector instructions plus any intervening rotations are grouped as possible main-steps.

4. Global connections suggest code boundaries. Thus, maximal sequences of visible vectors can be segmented on the basis of such connections.

This segmentation is tentative and may be revised in the light of later consideration of the model.

The segmentation advice derived from the annotation of TREE consists of three suggestions:

```
(SUGGESTION  (PURPOSE (TREE 10) MAIN-STEP)
             (REASON SUB-PROCEDURE CALL))

(SUGGESTION  (PURPOSE (TREE 20, 30, 40) PREP-STEP)
             (REASON MAXIMAL-SEQ-INVISIBLE-VECTORS))

(SUGGESTION  (PURPOSE (TREE 50) MAIN-STEP)
             (REASON MAXIMAL-SEQ-VISIBLE-VECTORS))
```

Reasons are given for the suggestions, allowing later analysis to have a basis for accepting or rejecting the suggestion. The reason could be USER-ADVICE, thus providing the ability for the user to interact with the system's reasoning process.

6.7 DEBUGGING ADVICE

Oddities in the form of the program can create a suspicion of bugs. The annotator notices these violations using Rational Form Criteria which are sensitive to unexpected and apparently erroneous code. Caveat comments are generated describing these complaints. The use of caveats was described in section 3.8.3 for the purpose of guiding the debugger. Additional examples of the use of caveat comments is found in the wishingwell debugging scenario of appendix C.

Rational Form Criteria are based upon expectations of simple efficiency and consist of noting sequences of the same turtle command. For example, it would be surprising to find a series of FORWARD instructions one after the other rather than a single command with the total movement expressed as the input. (Rational Form Criteria are also used by the State Editor, section 3.9, to tidy up the insertion of additional code by the debugger.)

6.8 ANALYSIS OF MORE COMPLEX PROGRAMS

This section explores possible extensions of the Annotator for the purpose of describing more complex programs.

6.8.1 Structural Analysis

Structural analysis observes the definition of the procedure rather than the process stack generated by running the program. For simple iterative and recursive procedures, structural analysis serves the important function of abstracting the basic round and the control structure. The round is sub-routinized and described in private. The control is specified in terms of the counter, step function, initialization, and exit condition of the loop or recursion. An integer arithmetic expert for simple control allows the system to predict the number of rounds and perceive such typical bugs as forgetting the exit condition or slipping through the end test.

A natural extension of annotation for turtle programs is to use structural analysis to recognize code for arcs. There is not a primitive ARC command in the LOGO turtle vocabulary. However, user-procedures for arcs are usually easy to recognize, appearing as loops of repeated small equal vectors interspersed by small equal rotations. The following turtle program is a typical arc procedure.

```
TO ARC :SIDE :DEGREES
10 IF :DEGREES<1 THEN STOP
20 RIGHT 1
30 FORWARD :SIDE
40 ARC :SIDE :DEGREES-1
END
```

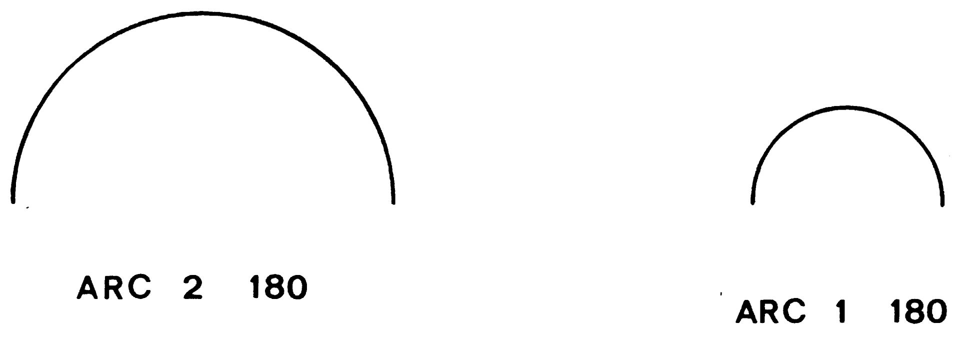


FIGURE 6.4

If semantics are provided for recognizing arcs and describing their important properties of radius and central angle, then this structural abstraction allows the remainder of the plan-finding and debugging system to treat arcs as simply another primitive. Using schematic analysis to recognize standard patterns for control or arcs requires that it precede the Process Annotation. In this way, the Process Annotation is advised of the important state descriptors for the abstracted code and does not produce unnecessary low-level assertions describing each vector and rotation in the arc.

To handle programs with inputs, it is important to describe the purpose of each variable. For example, in SCALED.TRI, structural analysis of the definition is useful to reveal that the input is serving the role of a scale factor. This allows the conclusion that the program draws a class of similar figures and that the input does not effect the shape.

```
TO SCALED.TRI :SCALE
10 FORWARD 100*:SCALE
20 RIGHT 120
30 FORWARD 100*:SCALE
40 RIGHT 120
50 FORWARD 100*:SCALE
END
```

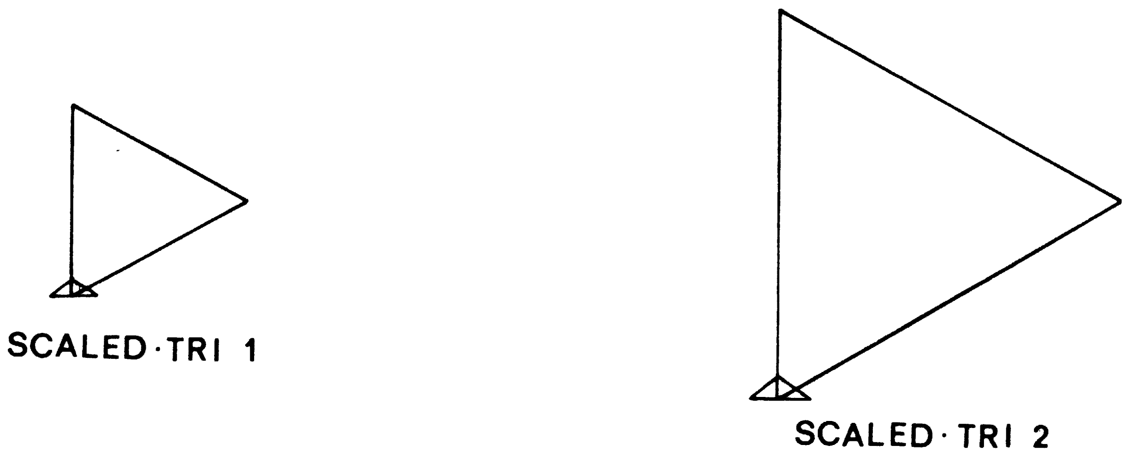


FIGURE 6.5

6.8.2 Formal Execution of Programs with Inputs

A mechanism for discovering schematic properties is formal execution of the procedure in which the program is run without binding variables to actual input data. The result is that the numerical coordinates of points are not known. However, certain facts are easy to determine. This would include the degrees of rotations or length of vectors where the input to the primitive is a constant. Similarly, local connections due to the sequential behavour of the turtle are revealed.

Global connection, however, can become arbitrarily difficult to ascertain without actually knowing the numerical coordinates. In formal execution, the binding process is simulated, allowing the equality of vectors because of identical formal inputs to be noted. (See [Hewitt 1972] on meta-evaluation.)

6.8.3 Schematic Versus Process Description

The process description readily identifies global connections wherein the turtle returns to a previous vertex. For fixed-instruction turtle programs, such occurrences are not dependent on the initial environment by the Rigid Body Theorem and are therefore recorded in the schematic description. For more complex programs with inputs, a problem arises in deciding which properties are schematic and which are dependent on the input binding. One mechanism for deciding this question is to run the program formally and solve the resulting analytic equations on coordinates; however, this rapidly becomes mathematically intractable. An alternative approach is to check the dependency of the program on the initial environment. If the dependency is solely for scale factor, global orientation or center of coordinates, then analysis reduces to the rigid body case. More generally, an alternative to formal analysis is to try a program with inputs on test cases. This leads to the problem of choosing representative cases, a good area for further research.

As programs become more complex, there is a trade-off between difficult to deduce but absolutely reliable schematic description and easy to obtain but questionable inductive properties. There is really no resolution: the important point is to be sensitive to the trade-off.

CHAPTER 7 -- FINDING THE PLAN

7.1 INTRODUCTION

Finding the plan is supplying the PURPOSE statements (<- commentary) illustrated in the previous examples for triangles, trees and stick figures. The model, the program and the performance annotation are all utilized to aid in this task. Plan-finding is needed when:

1. The user has supplied only minimal planning commentary.
2. The beginner is unable to express his plan.
3. A teacher has supplied the model and is analyzing students' programs, where the students' have not supplied any planning statements.

Given the model and program, the system begins by looking for a linear plan. The approach is to attempt to match model parts with modular main-steps and relations between model parts with preparatory-steps.

For an extended system, if the program were either iterative or recursive, then the system would look for a round plan. The goal would be to describe the generic element by the round and the relations between generic elements by the state interface between rounds.

Insertion or global plans are suggested by surprises in the syntactic structure of the program or suspicions about the cause of violations implied by the linear interpretation. For example, the occurrence of a sub-procedure in the midst of a code segment designed for one model part suggests a transition to accomplish a new model part. The old model part is presumably to be completed later. Demons are created for discovering when the system returns to finish the uncompleted part. If the intervening code is state-transparent, then the plan has an insertion structure. If the completion depends upon theorems about the performance of the intervening code, then the plan for the interrupted part is said to be global. Linear analysis coupled to an ability to debug this first-order approach in response to anomalies is a powerful reasoning mechanism which is used by the debugger as well.

Plan-finding obtains some guidance from the picture and some from the program in its effort to bind model parts to code. The picture supplies such clues as:

(a) global connections which suggest sub-picture boundaries;

(b) retracing which suggests inserts; and

(c) violations of model statements which are then used both as plausibility criteria (to distinguish between alternative interpretations) and to generate demons (which look for the completion of non-linear planning structures).

The program supplies quite different clues about intent. This includes:

(a) sub-procedure structure;

(b) surprises caused by one part being inserted into another;

both of which aid in recognizing sub-picture boundaries and

(c) the order in which the picture is drawn which, when combined with program-writing criteria, suggests the order in which the model parts are accomplished.

7.2 FINDING THE PLAN FOR STICKMAN

As an example, let us consider the problem of finding the plan for NAPOLEON. Recall that the procedure is:

```
TO NAPOLEON          ;See figure 1.7
10 VEE
20 FORWARD 100
30 VEE
40 FORWARD 100
50 LEFT 90
60 TRICORN
END
```

Assume that the VEE sub-procedure has been previously annotated and associated with the V model but that TRICORN and NAPOLEON have just been defined and their purpose is unknown. By considering sub-procedures as candidates for accomplishing model parts (analysis by synthesis), TRICORN is bound to the EQUITRI model. The result is two possible initial partial plans. These are:

```
PARTIAL.PLAN.1:                    PARTIAL.PLAN.2:
10 VEE <- (accomplish legs)        10 VEE <- (accomplish arms)
30 VEE <- (accomplish arms)        30 VEE <- (accomplish legs)
60 TRICORN <- (accomplish head)    60 TRICORN <- (accomplish head)
```

Further constraints are imposed by FINDPLAN's program-writing expectations. On the basis of BELOW, FINDPLAN expects:

```
(accomplish legs) <-> (accomplish arms) <-> (accomplish head)
```

The double arrow "<->" indicates that the sequence may happen in either forward or reverse order. On the basis of connectivity, the expectations are:

```
(accomplish legs) <-> (accomplish body) <-> (accomplish head)
```

Taken together, the result is that statement 10 is believed to accomplish the LEGS and statement 30 the ARMS. Thus, PARTIAL.PLAN.1 is preferred.

The code of the program is then considered statement by statement. Statement 20 draws a vector and is therefore believed to be the BODY. It might be only a piece of the body but this is not pursued until the linear assumption that the body is accomplished by a modular main-step is rejected.

Statements 30 and 60 have already been assigned to the arms and head, respectively. As a result, all of the model parts have been assigned but statement 40 remains unexplained. FINDPLAN consequently backtracks and interprets statement 20 as only a piece of the body. A demon is created for recognizing the body's completion and plan-finding recommences at statement 30. Statement 40 satisfies this demon since it draws a vector that begins at the endpoint of the first piece of the body. The result is that it is considered (piece 2 body). Thus, with almost no search, the plan for NAPOLEON is correctly deduced.

```
TO NAPOLEON          <- (accomplish man)
10 VEE               <- (accomplish legs)
20 FORWARD 100       <- (accomplish (piece 1 body))
30 VEE               <- (insert arms body)
40 FORWARD 100       <- (accomplish (piece 2 body))
50 LEFT 90           <- (setup heading)
60 TRICORN           <- (accomplish head)
END
```

7.3 PLAN-FINDING AS SEARCH

Finding the plan can be conceptualized as a search of a space of "partial plans". The search begins with the model, the program and the performance annotation. A partial plan is an explanation of some fraction of the model in terms of the program. Given a partial plan, its daughters are the result of generating alternative explanations for one of the remaining unassigned model parts. A terminal node is reached when all of the model parts have been explained and a complete plan is a path from the root to a terminal node, wherein an explanation is provided for how each model part is achieved.

A partial plan consists of PURPOSE comments which assign model predicates to code, unassigned model parts, expectations, the implied partial interpretation, and demons.

PURPOSES - These are the basic statements of a plan and appear as "<-" commentary in the NAPOLEON procedure. Five kinds of purposes are generated by FINDPLAN: accomplish, insert, setup, cleanup and retrace.

UNASSIGNED MODEL PARTS - The model specifies a list of parts. These are either primitive picture objects (vectors or rotations) or sub-models. An unassigned part is one without a PURPOSE statement indicating how it is to be accomplished.

EXPECTATIONS - These are predictions of which part is expected to be accomplished by the next main-step. They are based on applying program-writing criteria of efficiency and simplicity to the model.

PARTIAL INTERPRETATION - Model predicates can be evaluated by ordinary Cartesian geometry using the binding of model parts to code (which the plan implies) and an annotated description of the code's effects. A partial interpretation consists of those model predicates whose truth value is known given the current partial interpretation.

DEMONS - Demons are used to explain subsequent code in such a way that violations in the partial interpretation are eliminated. The elimination results from debugging the system's linear analysis and recognizing the existence of an interrupted or inserted main-step.

The partial plan is complete when all of the unassigned parts are explained by PURPOSES. Debugging is fixing the violations of the resulting complete interpretation.

7.4 LINEAR PLAN SPACE

The search for the plan is neither a standard breadth nor depth first exploration of the space. Instead, the system initially assumes a linear structure to the user's plan, looking to assign the parts to sequential code segments. The possibility that a part is being accomplished by disjoint segments of code or by insertions is not considered. This greatly constrains the search space. Branching, however, is not eliminated: for a given program, more than one linear plan will usually be possible. To choose among the alternatives in this linear plan space, several plausibility criteria are used consisting of (1) advice from the annotator and debugger, (2) expectations based on program writing criteria and (3) the number of violations implied by the partial plan. These are described in detail in the following paragraphs.

7.4.1 Advice from the User, Annotator and Debugger

Advice from the user, annotator or debugger is used to initialize the partial plan space. Annotator advice originates in noticing (1) subprocedures that have been previously associated with a model and (2) code groups that appear to have a common purpose on the basis of such non-model clues as subprocedurization, penstate changes and retracing. (These clues were listed in section 6.6.) The first produces PURPOSE assertions which form the initial partial plan; the second SUGGESTIONS which constrain the code group to being interpreted as either a main- or preparatory-step. Debugging advice is in the form of a request that the plan-finder supply a new plan that does not make certain hypotheses about the program. This interaction arises when the debugger finds all editing strategies for the current plan implausible.

7.4.2 Expectations Based on the Model

Another method for guessing the plan is to consider the model from the point of view of program-writing. Criteria of "efficiency" and "simplicity" suggest the order in which the parts will be accomplished. For example, it is expected that maximal advantage will be taken of the connections generated by the turtle's local behavior. Similarly, transitive sequences of predicates like ABOVE are expected to be accomplished in a way that minimizes the need to retrace.

The model is analyzed for transitive sequences of connectivity or relative position (ABOVE, BELOW, RIGHT-OF, LEFT-OF). The expectation is that that these sequences represent the probable order in which the parts are accomplished. For example, the MAN model suggests the transitive sequence LEGS -> ARMS -> HEAD (or vice versa) on the basis of both connectivity and position.

For sub-procedures or open-coded sequences, the expectations arise from analysis of the sub-model describing the current part being accomplished. This is simply a recursive analysis by the system which eventually terminates on primitive picture objects, i.e. rotations and vectors. For example, deciding that TRICORN in NAPOLEON accomplishes the EQUITRI model leads to the expectation of a sequence of equal vectors interspersed with 120 degree rotations. When the open-coded sequence is completed, then the system "pop"s back to the NAPOLEON model.

One effect of expectations is to suggest which partial plans be investigated first. For example, in considering an open-coded sequence for a triangle, each FORWARD might be an entry into an open-coded state-transparent sub-procedure for some other part. However, although this possibility does exist, it is not "expected" and is consequently not explored until the more likely planning structures are considered. Thus, expectations provide a measure of plausibility which originates in a knowledge of planning. Naturally, if the plan suggested by the expectations leads to many violations, then alternative partial plans are explored. This search constraint is of an "antecedent" character in that it does not require comparison of alternative branches but rather directly suggests the preferred next step in the current plan.

7.4.3 Violations as Plausibility Criteria

A third method is a plausibility estimate of partial plans based on the number of violations implied by the binding of code to parts. These violations may have two causes:

(1) An error in the plan-finder's theory of the user's intent;
(2) An error in the user's implementation of his plan.

If the user's program were correct, then partial plans which implied violations could be rejected immediately. Of course, the program may not be bug free. In this case, the preferred plan is the one that implies the fewest violations. This will usually be the plan intended by the user, but this is not inevitable. If the program has many bugs, then the plan-finder may indeed be hopelessly confused with no way to judge alternative interpretations.

This is to be expected. A human programmer when given another person's code and a description of the task will find it more and more difficult to understand the program as the number of bugs increases.

Specifically, the estimate used is simply the number of satisfied model statements and expections minus the number of violated model statements and expectations. If the program is bug free and the plan is correct, then the plausibility number will be maximal. At any instant in time, only those plans with the highest plausibility number are explored. After analyzing a statement of code, the plausibility number is recomputed and the active plans are rechosen. Inactive plans are "hung" and are not resumed unless their active brethren become less plausible.

7.5 NON-LINEAR PLANS AND SELF-CRITICISM

Searching the linear plan space is not adequate to recognize inserted or interrupted main-steps. The basic mechanism for doing this is to generate demons at points where FINDPLAN becomes suspicious of its linear interpretation of the procedure. These demons await confirming evidence that the linear plan is incorrect and, upon such confirmation, modify the partial plan appropriately.

Suspicions arise when violations occur. The suspicion is that the violation is due, not to a bug in the user's program, but rather to an error in the plan-finder's theory of the user's intentions. The assumption of linearity is questioned. For example, violations of "equal length" or "connectivity" may be due to making the erroneous linear assumption that the offending part is achieved in a single segment of code. The violations are due to assuming that a piece is the entire part. Such a suspicion gives rise to a planning demon looking for the completion of the side in such a way that the violations are satisfied. It is in this way that insertion or global plans for a given part are recognized.

Demons are procedures composed of an activation pattern and a body. The demon sits on the sidelines ordinarily invisible to the plan-finding process until a statement of code is examined that satisfies the demon's activation pattern. At that point, the demon is invoked, grabs control from the plan-finder and executes its body. This use of demons is based upon an approach to comprehending children's stories developed in a recent thesis by Charniak [Charniak 1972]. Demons are required because the suspicion may be incorrect. The demon takes no action until the suspicion is confirmed by the occurrence of code that can naturally be interpreted as the completion of the interrupted part. If that code never occurs, then the demon does nothing and the linear interpretation is preferred.

To be precise, when FINDPLAN binds an unassigned model part M to a segment of code C and the resulting interpretation implies model violations, there are three possible explanations:

1. The code is in error: a bug has been discovered.
2. C is not intended to accomplish M. Choose another interpretation for C.

3. C accomplishes only a PIECE of M. The remainder of M is achieved in pieces.

Possibility 1 requires no special action by FINDPLAN: the violation will eventually be passed to DEBUG for correction. Possibility 2 requires that the a different linear plan be chosen. This will occur if the current linear plan becomes less plausible than alternative linear interpretations when compared in terms of the static plausibility function described earlier. Possibility 3, however, represents an error in the plan-finder's linear analysis of the program. Hence, to take account of possibility 3, demons are generated. These demons are looking for better interpretations than the current linear plan (i.e. interpretations which do not imply as many violations).

Suppose FINDPLAN has just decided that statement C achieves model part M and that this results in a violation because M is too small. FINDPLAN suspects that M may be being accomplished in pieces. A completion demon is created looking for subsequent code CC which would eliminate the violation if CC is interpreted as another PIECE of M. If such code is found, the action of the demon is to edit the original partial plan so that M is now considered as being achieved by an interrupted main-step. If the code between the pieces of the main-step returns the turtle to the exit state of the first piece, then it is interpreted as being an insertion. Completion demons are also created when a vector is too short to accomplish an intended connection. An example occurs in the linear interpretation of TRICORN (with corrected rotations) shown below:

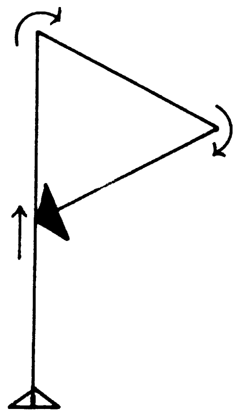
```
TO TRICORN       ;Incorrect linear plan initially deduced.
10 FORWARD 50   <- (accomplish (side 1))
20 RIGHT 120    <- (accomplish (rotation 1))
30 FORWARD 100  <- (accomplish (side 2))

  ;At this point in the plan-finding process, the violation
  ;of unequal sides occurs.  A completion demon is created
  ;that is looking for a vector of length 50 that could be
  ;interpreted as the remainder of (side 1).

40 RIGHT 120    <- (accomplish (rotation 2))
50 FORWARD 100  <- (accomplish (side 3))

   ;Here the violation of (side 1) not being connected to
   ;(side 3) occurs.  A second completion demon is created
   ;that is looking for another PIECE of (side 1) that connects
   ;to (side 3).

60 RIGHT 120    <- (accomplish (rotation 3))
70 FORWARD 50   <- (accomplish ?)
END
```

Both of the completion demons are triggered by statement 70. The result is that statement 10 is reinterpreted to accomplish only (piece 1 (side 1)) and statement 70 is assigned the purpose of accomplishing (piece 2 (side 1)). This produces the correct plan. (If the original TRICORN had been used, the completion demon generated by statement 10 being too short would still have found the correct plan: the connection demon, however, would not have been of much help since the picture drawn by the buggy tricorn is not closed.)

Suspicions of some violations give rise to multi-purpose-code demons. For example, if a side of a square causes the violation of unequal sides because it is too long, then the suspicion arises that only part of the long vector is intended to be the side. The remainder serves some other purpose. A demon is created looking for confirmation that the long vector serves multiple purposes. This confirmation is the subsequent occurrence of missing parts or completion demons (which are themselves created by other suspicions or surprises).

An example occurs in finding the plan for the following flag program and model:

```
MODEL FLAG
M1 PARTS BANNER POLE
M2 LINE BANNER
M3 TRIANGLE POLE
M4 CONNECTED POLE BANNER (VIA ENDPOINTS)
END

TO FLAG1
10 FORWARD 200  ;pole and side of banner
20 RIGHT 120
30 FORWARD 100  ;second side of banner
40 RIGHT 120
50 FORWARD 100  ;third side of banner
END
```



FLAG

FIGURE 7.1

Statement 10 is intended as both the the flagpole and a side of the banner. Initially, the linear attack interprets this vector as either the pole or a side of the banner, not both. However, the violation of "unequal sides" (which occurs under the latter interpretation) leads to a multi-purpose-code demon interested in finding another purpose for part of statement 10. Assigning to statement 10 the additional purpose of accomplishing the POLE satisfies this demon. The global connection in the interior of the vector drawn by statement 10 (which was noted by the annotator) confirms this interpretation and suggests to the plan-finder that the sub-vectors represent the division of labor.

If the model for the flag described the picture in terms of a LONG.POLE and a VEE, then the suspicion would not occur. Suspicions (and plans in general) are not absolute but relative to the description of the intent.

SUSPICION PATTERNS

Create a "completion demon" if the binding implies that a vector, rotation or arc is too small or a connection is not met. The demon is to be invoked by the occurrence of a statement of code which eliminates the originating violation. The action of the demon is to sprout a partial plan wherein the purpose of the statement is described as "(PIECE I <originating object>)".

Create a "multi-purpose-code demon" if the binding implies that a vector, rotation or arc is too large. The demon is to be invoked by any missing parts. Its action is to explain the missing part as a second purpose of the originating code.

7.6 SURPRISE ANALYSIS

A surprise refers to the encounter of unexpected statements inside the open-coded sequence for some model part. They are used to hypothesize an interrupted main-step, without actually going to the effort of pursuing an incorrect linear interpretation and correcting oneself on the basis of suspicions. Essentially, they represent the alternative approach of guessing immediately what the unexpected code means rather than waiting for suspicious model violations to occur. The question is when to treat unexpected code as a surprise and when to be more cautious and generate only a suspicion demon while remaining within a linear plan.

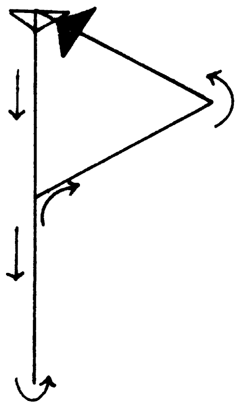
The criterion used is whether the unexpected code represents a transition in syntactic type, for example, from a sequence of primitives to a subprocedure. In this case, the anomaly is treated as a surprise. If however, the unexpected code is simply an unexpected primitive than the probability that this may be only a bug in a linear plan is too great to immediately reject the linear interpretation. In this latter case, the suspicion mechanisms described previously are used.

As an example of surprise analysis, consider the following FLAG2 procedure in which the POLE is accomplished by being inserted into an open-coded sequence for the BANNER.

```
TO FLAG2
10 FORWARD 100  ;First side of the banner.
20 POLE
30 LEFT 120
40 FORWARD 100  ;Second side of the banner.
50 LEFT 120
60 FORWARD 100  ;Third side of the banner.
END

TO POLE         ;State-transparent insert for the pole.
10 FORWARD 100
20 BACK 100
END
```

The decision that an equilateral triangle is being open-coded produces the expectations of a linear sequence of equal vectors and 120 degree rotations. These expectations arise from an examination of the EQUITRI model. The POLE subprocedure then is a surprise because it occurs in the midst of expecting a triangle. The surprise of an unexpected statement of code that violates the current expectation can be explained in two ways:

1. The intention was to satisfy the expectation and the statement



FLAG

FIGURE 7.2

is in error, e.g. statement 20 was intended to be RIGHT 120 and the POLE subprocedure is out of place.

2. The expectation is inappropriate because the plan-finder's linear theory of the user's plan was in error. The statement is not part of the open-coding of the current model part. Rather, the code is the beginning of another part. The planning structure may be that of an insertion of the new part or it may be that the old part is to be completed by global effects.

Surprise analysis pursues the latter hypothesis. Statement 20 is interpreted as a new model part: in this case the POLE. A demon is created whose purpose is to note code that may be the completion of the interrupted sequence. Otherwise, the completion may be misinterpreted as part of the basic linear planning sequence.

In FLAG2, the code for the triangle is interrupted by the insert for the POLE. The result is that a demon is created awaiting completion of the insert and recommencement of the interrupted part. Specifically, the demon is looking for a RIGHT 120. Consequently, the demon is activated by statement 30 with the result that this statement is assigned the purpose of being "(ROTATION 2 BANNER)".

7.7 TREE1, AN ILLUSTRATION OF HETERARCHY IN PLAN-FINDING

This section provides an example of recursive interaction between the plan-finder and the debugger. The TREE1 procedure with its associated model and the correct plan is shown below:

```
MODEL TREE
M1 PARTS TOP TRUNK
M2 LINE TRUNK
M3 EQUITRI TOP
M4 VERTICAL TRUNK
M5 COMPLETELY-BELOW TRUNK TOP
M6 CONNECTED TOP TRUNK
M7 HORIZONTAL (BOTTOM (SIDE TOP))
END
```

```
TO TREE1
10 TRIANGLE      <- (accomplish top)
20 RIGHT 50      <- (setup heading)
30 FORWARD 50    <- (retrace)
40 RIGHT 50      <- (setup heading)
50 FORWARD 100   <- (accomplish trunk)
END
```

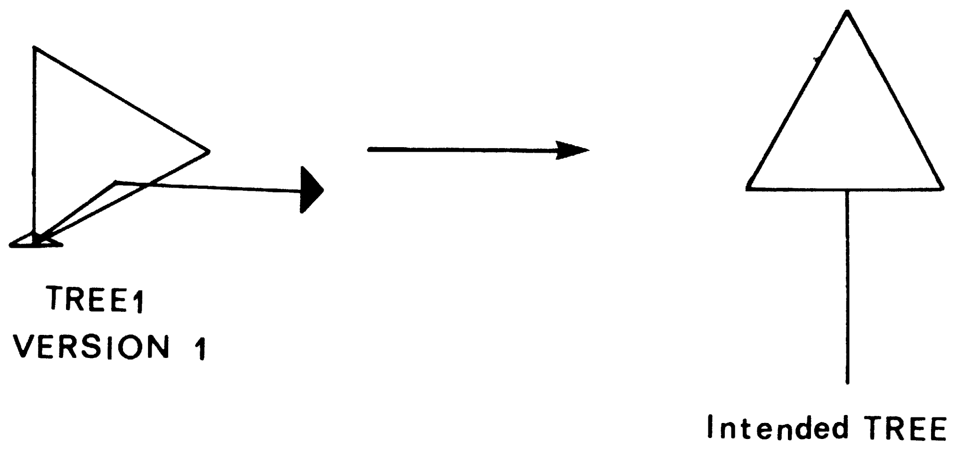


FIGURE 7.3

In TREE1, the rotations between the TOP and the TRUNK in statements 20 and 40 are incorrect. (See section 3.2 for a Debugging Scenario for this program.) Segmentation advice supplied by the annotator initially misleads the plan-finder into the belief that the TRUNK is accomplished by both statements 30 and 50. The result is that the plan-finder first produces the following linear plan which errs in its interpretation of the TRUNK.

```
TO TREE1         <- (accomplish tree)
10 TRIANGLE      <- (accomplish top)
20 RIGHT 50      <- (setup heading)
(30-50) LINE     <- (open-coded sequence for trunk)
END


TO LINE          <- (accomplish line)
30 FORWARD 50    <- (accomplish (piece 1 line))
40 RIGHT 50      <- (setup heading)
50 FORWARD 100   <- (accomplish (piece 2 line))
END
```

FINDPLAN identifies statement 10 as the TOP of the tree by the fact that the user-subprocedure TRIANGLE has been previously identified with the EQUITRI model. One of the definitions of a line is that it is a set of collinear vectors. Statements 30, 40 and 50 satisfy the requirement that the vectors be connected although they are not parallel. The interpretation that these vectors are the TRUNK is selected by the plan-finder as the best choice. Alternative interpretations imply more violations. (For example, the decision that statement 50 is the TRUNK has the unsatisfactory consequence that statement 30 is an unexplained visible vector.) As the following discussion indicates, the system eventually discovers its mistake and finds the correct plan.

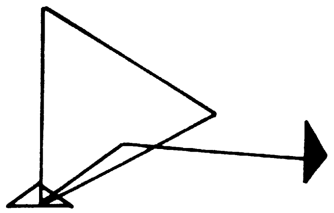
The plan, procedure and model are given to the debugger to correct the implied violations. The debugger is, however, unable to repair the program satisfactorily, given the commitment that statements 30 and 50 are together the TRUNK. No debugging strategies that will make the TRUNK into a LINE are found plausible.

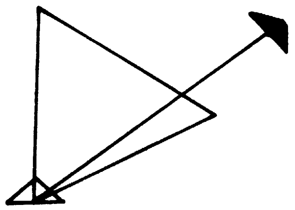
1. One strategy analyzed by the Debugger is to make the body vectors collinear by altering the intervening rotation of statement 40 to be zero degrees. On the basis of rational form criteria, statement 40 becomes pointless and is deleted, producing figure 7.4. The Debugger is very reluctant to delete completely a statement of the user's code and therefore seeks an alternative strategy.

2. Another way to make the body vectors parallel is to alter statement 40 to be "RIGHT 180". This produces figure 7.5. The result is that the TRUNK now overlaps the TOP of the tree in such a way that the two violations of (VERTICAL TRUNK) and (COMPLETELY-BELOW TRUNK TOP) cannot both be fixed. Fixing one unavoidably violates the other.

USING PLAN 1 AS GUIDANCE FOR FIXING

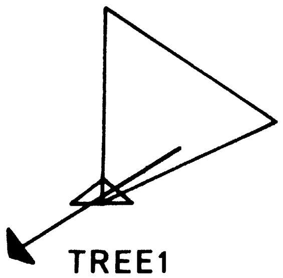
TREE1





TREE1
VERSION 2A

FIGURE 7.4



TREE1
VERSION 2B

FIGURE 7.5

MAKING LINES 30 AND 50 INTO A LINE

When no plausible debugging solution is found, the system reinvokes the plan-finder with a request for a new plan. Along with this request, the Debugger informs the plan-finder of that part of the current plan which was found unsatisfactory. In this particular case, the system asks for a new plan which provides an alternative binding for TRUNK. Thus, the control structure is not a strict hierarchy with one-way communication between stages but a heterarchy with two-way communication occurring between stages of the analysis process. (The notion of heterarchy is introduced in [Minsky 1972].)

The plan-finder backtracks and selects the next most plausible plan using the advice that the TRUNK is not accomplished by the combined action of statements 30 and 50. The two possibilities are that either statement 30 or statement 50 is the TRUNK. The former implies an unexplained part (statement 50) which is particularly implausible and therefore the latter is preferred. Hence, the result is a plan wherein statement 30 is explained as an interface (with a visibility bug) and statement 50 is the TRUNK.

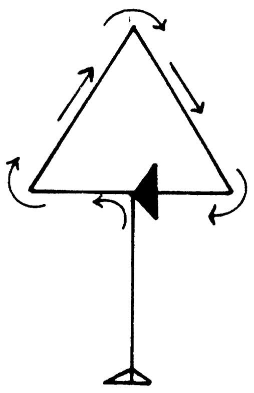
```
TO TREE1              <- (accomplish tree)
10 TRIANGLE           <- (accomplish top)
20 RIGHT 50           <- (setup heading for retrace)
30 FORWARD 50         <- (retrace)
40 RIGHT 50           <- (setup heading for trunk)
50 FORWARD 100        <- (accomplish trunk)
END
```

This is the correct plan. When given to the debugger, the program is successfully and plausibly edited to eliminate all of its bugs (section 3.2).

7.8 TREE2, AN OPEN CODED TREE PROGRAM

TREE2 is a linear, open-coded procedure for the TREE utilizing no retracing or insertions. It is included here as an example to indicate the simplicity of discovering the plan of open-coded procedures when they have no bugs and to illustrate precisely the partial plan data structure.

```
TO TREE2          ;See figure 7.6.
10 FORWARD 100    ;Statement 10 accomplishes the trunk.
20 LEFT 90
30 FORWARD 50     ;Statements 30-90 accomplish the
40 RIGHT 120      ;triangle by starting in the
50 FORWARD 100    ;middle of the bottom side.
60 RIGHT 120
70 FORWARD 100
80 RIGHT 120
90 FORWARD 50
END
```



TREE2

FIGURE 7.6

The need to recognize open-coding greatly increases the number of possible planning interpretations. Open-coding is the opposite of subroutinization and is defined by code for a common purpose appearing directly in the procedure rather than being compartmentalized into a subprocedure definition. Open-coding is recognized by recursing the planfinder and attempting to determine if the part of a given sub-model is being achieved by the current statement. In TREE2, this means that individual vector instructions may be interpreted as accomplishing the subparts (sides) of a part (top) of the tree.

This recursion can result in considering whether a part of a part of a sub-model is being attempted. Thus it is clear that, as the complexity of the model increases, open-coding becomes more and more difficult to recognize by the plan-finder and debug by the user. If subroutinization is used, plan-finding does not become more complex with increasing model complexity (providing, of course, that the subroutinization corresponds in some reasonable way to the model parts). This is to be expected: understanding many statements of open-code with bugs and with no segmentation clues is very difficult for people.

Because there are no bugs, anotator advice serves to make the problem of finding the plan for TREE2 manageable. The global connection between the vectors drawn by statements 10, 30 and 90 causes the annotator to suggest a code boundary at this point. This implies to the plan-finder that the procedure is composed of two main parts: one consisting of statement 10 and the other of statements 30 through 90. This annotation advice prevents the plan-finder from exploring the possibility that statement 10 is only the beginning of an open-coded sequence for either the TRUNK or the TOP. Instead, statement 10 by itself must be the code for either the TOP or the TRUNK.

To be precise, the following two partial plans are produced in analyzing the purpose of statement 10: The -> symbol used in the expectation part of these partial plans designates the order in which the parts are expected.

```
PARTIAL-PLAN 1 (PROGRAM TREE2) (MODEL TREE)
 PURPOSES:
  P1 ((PURPOSE (TREE2 10) (ACCOMPLISH TRIANGLE)
      (REASON: OPEN-CODING OF TRIANGLE TO ACCOMPLISH TOP))

 UNASSIGNED PARTS: TRUNK
 EXPECTATIONS:
  FROM TREE MODEL, TOP -> TRUNK
 INTERPRETATION:
  SATISFACTIONS: NONE
  VIOLATIONS: (WRONG NUMBER OF SIDES), (NO ROTATIONS), etc.

PARTIAL-PLAN 2 (PROGRAM TREE2) (MODEL TREE)
 PURPOSES:
  P1 ((PURPOSE (TREE2 10) (ACCOMPLISH TRUNK))
      (REASON: OPEN-CODING OF LINE TO ACCOMPLISH TRUNK))

 UNASSIGNED PARTS: TOP
 EXPECTATIONS: FROM TREE MODEL, TRUNK -> TOP
 INTERPRETATION:
  SATISFACTIONS: (LINE TRUNK)
  VIOLATIONS: NONE
```

Naturally, there are fewer violations when statement 10 is considered a LINE rather than the entire TRIANGLE. Consequently, partial plan 2 is preferred in which statement 10 is correctly analyzed as accomplishing the TRUNK.

Suspicions cause statements 30-90 to be correctly interpreted.

```
TO TREE2       <- partial-plan 2
10 FORWARD 100 <- (accomplish trunk)
20 LEFT 90

;FINDPLAN about to consider statement 30.
30 FORWARD 50  ;Statements 30-90 accomplish the
40 RIGHT 120   ;triangle by starting in the
50 FORWARD 100 ;middle of the bottom side.
60 RIGHT 120
70 FORWARD 100
80 RIGHT 120
90 FORWARD 50
END
```

Linear analysis will assign statement 30 to accomplish the first side and statement 50 to accomplish the second side of the top. This results in a violation of unequal sides. As a result, a suspicion demon is created looking for the completion of the first side. To be precise, it is looking for a vector of length 50 which is collinear with the vector drawn by statement 30. Statement 90 satisfies this demon with the result that the first side is interpreted as occuring globally in two parts and the violation of the sides being of unequal length is eliminated. The end result is that (side 1) is recognized as being accomplished by an interrupted main-step. This results in TREE2 being assigned the following plan:

```
(FINDPLAN (PROGRAM TREE2) (MODEL TREE))

TO TREE2           <- (accomplish tree)
10 FORWARD 100     <- (accomplish trunk)
20 LEFT 90         <- (setup heading)
(30-90) TRIANGLE   <- (open-coded sequence for top)
END


TO TRIANGLE     <- (accomplish triangle)
30 FORWARD 50   <- (accomplish (piece 1 (side 1 triangle)))
40 RIGHT 120    <- (accomplish (rotation 1 triangle))
50 FORWARD 100  <- (accomplish (side 2 triangle))
60 RIGHT 120    <- (accomplish (rotation 2 triangle))
70 FORWARD 100  <- (accomplish (side 3 triangle))
80 RIGHT 120    <- (accomplish (rotation 3 triangle))
90 FORWARD 50   <- (accomplish (piece 2 (side 1 triangle)))
END
```

Without the advice from the annotator that statement 10 is a main-step, the plan-finder must consider the possibility that it is the beginning of an open-coded segment for the top of the tree. This possibility along with the correct interpretation that statement 10 is the trunk is shown in figure 7.7. The plan-finder would still be successful. In fact, even if all of the rotations were incorrectly 90 degrees, the search would still eventually deduce the right plan. This is illustrated in the following paragraphs.

BUGGY.TREE2 is intended to be the correct open-coded tree program TREE2; however, the triangle rotations are erroneously 90 degrees.

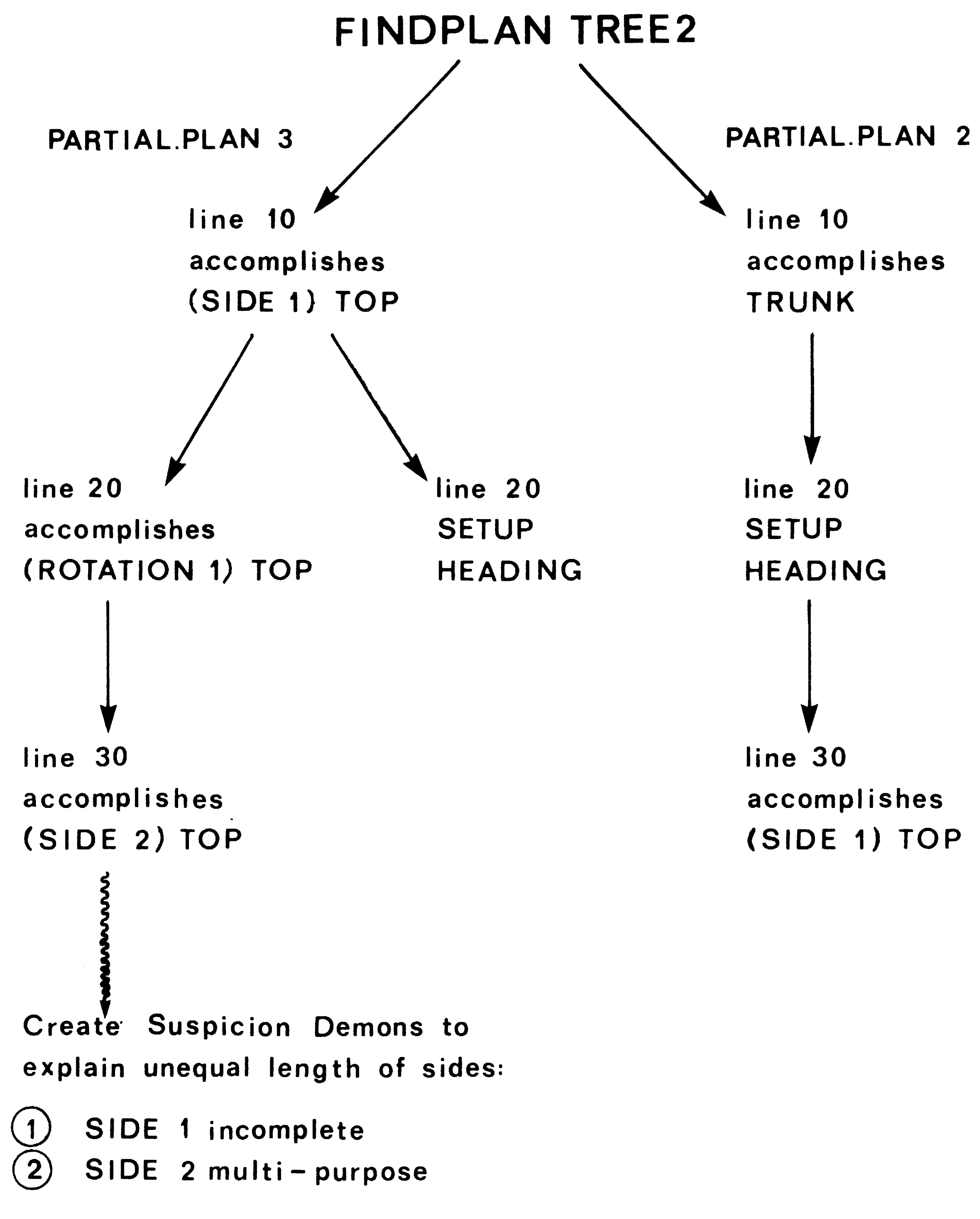
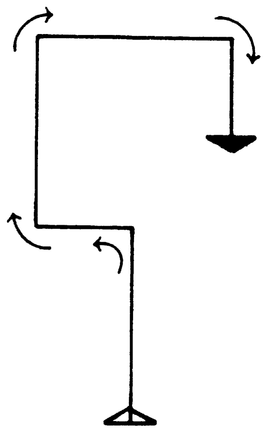

FINDPLAN TREE2
PARTIAL.PLAN 3
PARTIAL.PLAN 2
line 10 accomplishes (SIDE 1) TOP
line 10 accomplishes TRUNK
line 20 accomplishes (ROTATION 1) TOP
line 20 SETUP HEADING
line 20 SETUP HEADING
line 30 accomplishes (SIDE 2) TOP
line 30 accomplishes (SIDE 1) TOP
Create Suspicion Demons to explain unequal length of sides:
1 SIDE 1 incomplete
2 SIDE 2 multi-purpose

FIGURE 7.7

```
TO BUGGY.TREE2
10 FORWARD 100   ;the trunk
20 LEFT 90
30 FORWARD 50    ;part 1 of side 1 of the top
40 RIGHT 90      ;bug: should be RIGHT 120
50 FORWARD 100   ;side 2 of the top
60 RIGHT 90      ;bug: should be RIGHT 120
70 FORWARD 100   ;side 3 of the top
80 RIGHT 90      ;bug: should be RIGHT 120
90 FORWARD 50    ;part 2 of side 1 of the top
END
```



BUGGY.TREE2

FIGURE 7.8

The result is that the annotator cannot supply any segmentation advice on the basis of global connections. Consequently, more possibilities are explored than for TREE2. Specifically, two more possibilities for the interpretation of statement 10 must be considered: that statement 10 accomplishes the first side of the triangle and that statement 10 accomplishes the first piece of the trunk.

```
PARTIAL-PLAN 3 (PROGRAM BUGGY.TREE2) (MODEL TREE)
 PURPOSES:
  P1 ((PURPOSE (TREE2 10) (ACCOMPLISH (SIDE 1 TRIANGLE))
      (REASON: BEGINS OPEN-CODING OF TRIANGLE TO ACCOMPLISH TOP))

 UNASSIGNED PARTS: TRUNK
 EXPECTATIONS:
  FROM TRIANGLE MODEL, ROTATION -> 2(SIDE -> ROTATION)
  FROM TREE MODEL, TOP -> TRUNK
 INTERPRETATION:
  SATISFACTIONS: NONE
  VIOLATIONS: NONE

PARTIAL-PLAN 4 (PROGRAM BUGGY-TREE2) (MODEL TREE)
 PURPOSES:
  P1 ((PURPOSE (TREE2 10) (ACCOMPLISH (PIECE 1 TRUNK)))
      (REASON: BEGINS OPEN-CODING OF LINE TO ACCOMPLISH TRUNK))

 UNASSIGNED PARTS: TOP
 EXPECTATIONS: FROM TREE MODEL, TRUNK -> TOP
 INTERPRETATION:
  SATISFACTIONS: NONE
  VIOLATIONS: NONE
```

Partial plans 2, 3 and 4 are equally plausible at this point. However, further analysis eventually results in partial plan 2, the correct plan, implying the fewest violations and therefore being chosen.

An interesting type of equivalence that one does not see in the literature on program verification is that of equivalence with respect to the plan. TREE2 and BUGGY.TREE2 are equivalent in this sense despite that fact that the latter fails completely to accomplish the intended TREE. This equivalence is interesting because it reflects an identity with respect to intent, not simply actual performance.

7.9 PLAN-FINDING FOR SORTING PROGRAMS

Greg Ruth in a recent thesis [Ruth 1973] explores the problem of diagnosing the bugs in simple sorting programs written by students. Just as in our turtle world, he has the problem of finding the plan for the student's program. His solution, however, is somewhat different.

As he observes, a model of a sort program for an array of integers A(1) , ... , A(n) is simply the statement that the items are in order, i.e.

A(i)>A(j) => i>j.

This does not provide much information for understanding the intricacies of the many varieties of sorting programs, e.g. bubble sorts, merge sorts, interchange sorts.

Therefore, Ruth's plan-finder is supplied with a description of common algorithms for sorting. The distinction between an algorithm and its implementation is that (1) the algorithm is written in a higher language and (2) the algorithm indicates various implementation choices and typical bugs. The first property of the algorithm specification is illustrated by the existence of an INTERCHANGE primitive in the algorithm language while several assignments statements and the creation of a temporary variable are required in the actual program.

```
        ALGORITHM                  PROGRAM

        (INTERCHANGE X Y)          (SETQ T X)
                                   (SETQ X Y)
                                   (SETQ Y T)
```

The second property of Ruth's algorithm specifications is illustrated by the fact that the description of a bubble sort would include the information that the sort could either bubble up the greatest element or the least element depending on whether the array is sorted from A(n) to A(1) or A(1) to A(n). A typical bug would be to slip through the endtest causing infinite repetition.

The notion of understanding common algorithms to aid in finding the plan for a particular program is found in MYCROFT as well. Recall that the plan-finder generates expectations from a consideration of the picture model from a program-writing standpoint. These expectations are quite similar to Ruth's algorithm descriptions. For example, from the MAN model, MYCROFT expects the limbs to be accomplished in the order:

LEGS <-> ARMS <-> HEAD.

For understanding more complex programs, it seems clear that knowledge based on the expected algorithm will play a role. One cannot expect a _program-understanding system_ to redevelop all of the clever algorithms that can be used to accomplish some goal. It is enough of a task to understand a particular implementation (possibly with bugs), given knowledge of these algorithms. On the other hand, the importance of research that investigates the transition from a model of the intended result to possible algorithms should not be underestimated. Such research is fundamental to the design of _program-writing systems_.

7.10 CONCLUSIONS

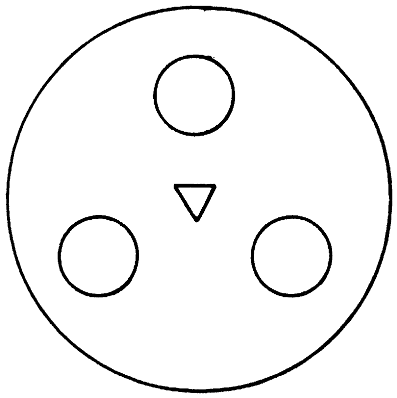
The algorithm for plan-finding performs well when:

(1) The user supplies advice in the form of a partial plan;
(2) The procedure has subroutines;
(3) The procedure has few bugs.

If the program is not subroutinized and is full of bugs, the search grows unmanageable and difficulties arise in selecting the most plausible candidate. This performance is quite reasonable in the sense that exactly the same statements are true of a human problem solver investigating a strange program.

It is possible that the user's intended plan cannot be discovered at all: bugs can make any of several alternative interpretations equally plausible. In figure 7.9, it is ambiguous which of the small circles of SYM.FACE were intended to be the eyes and which the mouth. In such cases, the system must ask advice as to the proper binding of code-segments to model parts. This is not unreasonable since the user certainly knows the intended binding.

The system is not prepared to find the clever plans of expert programmers, nor is it prepared to deal with situations where the program is



SYM.FACE

FIGURE 7.9

based on a plan which is not suggested by the model. This occurs when domain-dependent knowledge not mentioned in the model is used to justify global effects. An example is INSCRITRI, a triangle program based on the relationship between a triangle and its inscribed circle (figure 7.10). The triangle is drawn from a local home at the center of the triangle. The system does not know, unless it is asserted explicitly in the model, the geometric relationship between the radius of the inscribed circle and the side of the triangle.

Program writing leaves the system free to choose a plan for programming the model. Hence, it is never necessary to guess intent: the plan is not found, it is constructed. Thus, although knowledge of the form and implications of different plans is essential to program understanding, debugging and writing, the search problem of finding the plan, in ignorance of the user's intent, is not relevant to designing programs. Nevertheless, plan-finding remains important as a crucible for testing the effectiveness of the system's knowledge of plans, as a procedure for expanding sparse commentaries and as a mechanism for generating intelligent questions if aid must be

```
TO INSCRITRI :X
10 TEE :X
20 RT 120
30 TEE :X
40 RT 120
50 TEE :X
END

TO  TEE :X
5 MAKE "Y" :X * (SQRT 3.0)
10  FD :X
20  RT 90
30  FD :Y
40  BK 2 * :Y
50  FD :Y
60  LT 90
70  BK :X
END
```

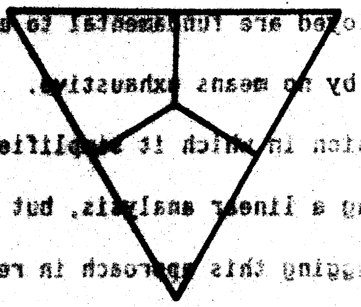


FIGURE 7.10

requested from the user.

CHAPTER 8 -- EXTENSIONS

8.1 RECAPITULATION

This paper has presented a system for annotating, planning and debugging simple fixed-instruction turtle programs. The methods and knowledge employed are fundamental to understanding more complex programs, though they are by no means exhaustive. An important characteristic of the system is the fashion in which it simplifies debugging the procedure and finding the plan by using a linear analysis, but is capable of handling more complex problems by debugging this approach in response to surprises and suspicions.

8.2 EXTENSION TO MORE COMPLEX PROGRAMS

An obvious extension is to repairing the bugs of more complex programs. The first step would be towards understanding round-structured programs. Following this, the analysis of programs with inputs and programs with more complex forms of iterative and recursive control should be undertaken.

An interesting step beyond the determinism of the display turtle would be to consider turtles with sensory input as, for example, might be provided by touch and light sensors. Here, the environment can hold surprises. It is not possible to use the complete world description that is possible in annotating the performance of ordinary turtles: the environment of a touch-sensitive turtle can contain unexpected obstacles. This extension is particulary interesting because the domain can be extended from the geometry of simple pictures to the behaviour of biological creatures. The sensors allow simulation of various animate activities of hiding, seeking, eating and mating [Goldstein 1973].

8.3 EXTENSION TO LEARNING

The monitor does not reason about the nature of the model. It does not attempt to note similarities between models, perform generalizations of symbolic models from sketches, or recognize new models in terms of older ones. One method to accomplish this would be to view the picture model as a net (figure 8.1) and perform various matching and comparisons activites. This approach is developed by Winston in his paper on Learning Structural Descriptions [Winston 1970].

Sketches are over- rather than under-determined. (Our models have generally been under-determined.) The symbolic models utilized so far provide a clear target for the learning process. The problem can be characterized as properly generalizing a sketch into a symbolic description. An important question is whether a more knowledge-based procedural approach is preferable to the uniform net matching of Winston.

MYCROFT is a performance model. It does not learn to plan, write or debug programs. A difficult but valuable extension would be to consider how such competence might be acquired. The organization of the system's analysis into an initially "linear" attack followed by debugging its explanations to handle more difficult situations could serve as a basis for self-improvement. Sussman discusses some methods for a program writing system to learn by debugging its own programs [Sussman 1973]. Much more research, however, is needed on this issue.

8.4 EXTENSIONS TO EDUCATION

An understanding of the knowledge needed to plan, describe, debug and write programs is important to developing an educational curriculum. The thesis of the LOGO project is that such abilities are fundamental to problem

NET REPRESENTATION OF A TREE MODEL

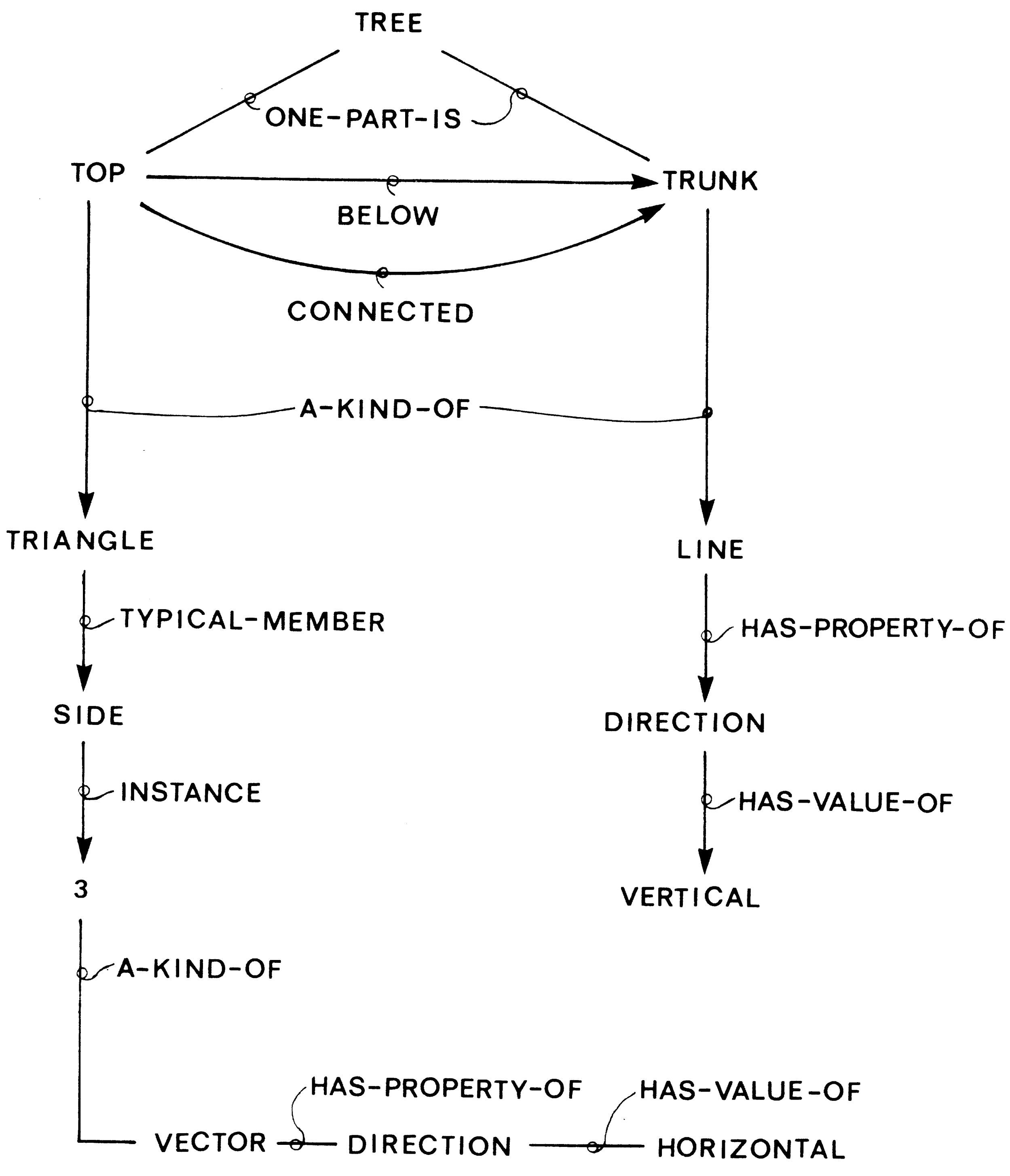


FIGURE 8.1

solving and thinking. The competence model developed in this paper makes precise some of the concepts that are at the foundation of procedural thinking. From an educational standpoint, this research has several possible applications.

It can serve as the basis of a monitor for aiding in the diagnosis and correction of errors. The system can not only diagnose and correct bugs, but can also display its analysis and inner thinking to the student; and thereby serve as a model of the debugging process.

An interesting extension might be to build an editor that had access to planning knowledge. A user could then modify (or write) his programs with high level assertions about the structure of the plan and the purposes of the code. The editor could translate such instructions into sub-routinization, proper state interfacing, and commentary to simplify later debugging.

The user might be asked to supply his plan in the planning vocabulary discussed in this thesis. By being asked to be explicit, he will acquire better problem solving habits. As the system can utilize such plans to aid in debugging, the extra burden will not seem pointless to the student.

Finally, perhaps the student himself might be asked to engage in the kind of research described in this thesis: namely the activity of making explicit basic concepts of planning, programming and debugging. The student would then be in a position to bootstrap his problem solving competence as well as design his own personal computer assistant to aid in the programming proce

APPENDIX A -- THE TURTLE LANGUAGE

A.1 THE STATE OF THE TURTLE

The state of the turtle consists of :HEADING, :POSITION, and :PEN. :HEADING is the direction in which the turtle is pointed and is a number between 0 and 360. A convention is that a heading of zero points towards the top of the page. :POSITION is the location of the turtle on the screen. The center of the screen is (0 0). :PEN is either "UP" (invisible) or "DOWN" (visible). The HOME state is:

```
:HEADING  = 0
:POSITION = (0 0)
:PEN      = "DOWN"
```

A.2 THE BASIC TURTLE PRIMITIVES

Turtle programs will be limited to the primitives:

FORWARD :R moves the turtle :R steps starting from its current position and in the direction of its current heading.

BACK :R = FORWARD -:R

RIGHT :A turns the turtle :A degrees clockwise. The new heading is the old heading plus :A. The position is not changed.

LEFT :A = RIGHT -:A

PENUP lifts the pen up, making subsequent movements leave no track.

PENDOWN places the pen down, making subsequent movements leave a trail.

These commands will usually be abbreviated.

FD	=	FORWARD
BK	=	BACK
RT	=	RIGHT
LT	=	LEFT
PU	=	PENUP
PD	=	PENDOWN

A.3 FIXED INSTRUCTION TURTLE PROGRAMS

The turtle programs which are analyzed are built from the six turtle primitives -- FORWARD, BACK, RIGHT, LEFT, PENUP, PENDOWN -- and the use of sub-procedures. Furthermore, they are restricted to be Fixed-Instruction. This excludes the the use of conditionals, iteration, recursion, variables, arrays and interrupts. The input to the turtle primitives consists of numerical constants. These programs must halt.

Such simple programs are of interest because one cannot hope to understand more complex programs without a foundation adequate to cope with simple procedures. Furthermore, these programs are far from trivial with respect to repairing their bugs. Finally, many of the ideas introduced generalize naturally to handle more complex programs.

A.4 BEYOND FIXED-INSTRUCTION PROGRAMS

The LOGO language itself is certainly not limited to fixed-instruction programs. Iteration, recursion, conditionals, variables and, in the most current PDP-11 version of the language, even interrupts are possible.

More importantly, although fixed-instruction programs are usually the first ones written by beginners, students are soon introduced to more complex procedures. A natural extension of the research described in this paper would be to build a system capable of understanding the bugs that occur in these evolutionary sequences.

To illustrate the more complex kinds of LOGO turtle programs that are written and also to provide an example of a typical evolutionary sequence, the following paragraphs illustrate the evolution of a POLY program capable of drawing an arbitrary regular polygon. The first step would be within MYCROFT's current competence: namely to understand and debug fixed-instruction programs for simple geometric shapes. The next would be to understand the problems that usually arise in generalizing such procedures to variable sides. An example of such a transition is shown below:

```
TO TRIANGLE  -->            TO TRIANGLE.WITH.INPUTS :SIDE
10 FORWARD 100                      10 FORWARD :SIDE
20 RIGHT 120                20 RIGHT 120
30 FORWARD 100                      30 FORWARD :SIDE
40 RIGHT 120                40 RIGHT 120
50 FORWARD 100                      50 FORWARD :SIDE
60 RIGHT 120                60 RIGHT 120
END                         END
```

Subsequent steps in this process of generalization would typically be to supply a control structure so that the procedure could draw a figure with an arbitrary number of sides and then couple that with a variable rotation between sides. The result is a POLY procedure which can draw any regular polygon.

```
TO POLY :SIDE :ANGLE
10 FORWARD :SIDE
20 RIGHT :ANGLE
30 POLY :SIDE :ANGLE
END
```

The student might then wish to add a counter in order that the program terminate. This is necessary if it is to be used as a sub-procedure. Designing a counter requires the use of conditionals and exiting commands.

```
TO STOP.POLY :SIDE :ANGLE :SIDES
10 IF :SIDES=0 THEN STOP
20 FORWARD :SIDE
30 RIGHT :ANGLE
40 STOP.POLY :SIDE :ANGLE :SIDES-1
END
```

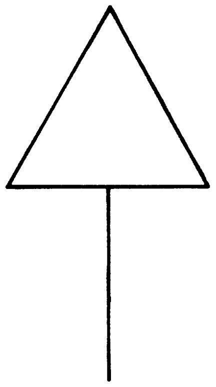
From a research standpoint, the goal is to tackle the difficult problem of understanding variables, conditionals, iteration and recursion by studying the development of [illegible] trying to analyze a complex program without [illegible] was written.

APPENDIX B -- PICTURE MODELS

This appendix describes in detail models for the TREE and TRIANGLE and then discusses general issues which arise in considering descriptions of turtle pictures.

B.1 THE TREE MODEL

```
MODEL TREE
M1 PARTS TOP TRUNK
M2 LINE TRUNK
M3 TRIANGLE TOP
M4 VERTICAL TRUNK
M5 BELOW TRUNK TOP
M6 CONNECTED TOP TRUNK
END
```



Intended TREE

FIGURE B.1

M1 PARTS TOP TRUNK

The PARTS statement provides names for sub-pictures. It is not strictly necessary and could be deduced by observing the inputs to the remainder of the model statements, but is useful for clarity.

M2 LINE TRUNK

(LINE TRUNK) requires TRUNK to be a line segment. Its semantics are:

(OR (VECTOR X) (AND (VECTORS X) (PARALLEL X) (CHAIN X))).

This definition is ordinarily used to verify the predicate with respect to the Cartesian picture. The debugging chapter discusses how these semantics can be used in an imperative way to repair sequences of vectors that are intended to be collinear but fail.

M3 TRIANGLE TOP

(TRIANGLE TOP) demands that the sub-picture TOP satisfy the model defined for TRIANGLE. This sub-model is discussed in section B.2. Thus models, once defined, become predicates in the picture language.

M4 VERTICAL TRUNK

The semantics for VERTICAL are:

```
(OR (= (DIRECTION X) 0) (= (DIRECTION X) 180))
```

where "X" must be bound to a vector. The Annotator records the DIRECTION of vectors as a heading between 0 and 359 degrees. See chapter 6.

M5 BELOW TRUNK TOP

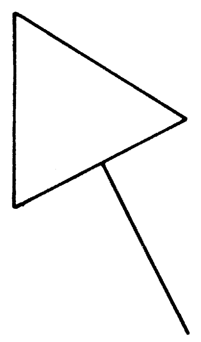
(BELOW X Y) can have several meanings, where X and Y are sub-pictures (sets of vectors).

PARTLY-BELOW	- some point of X is below Y
COMPLETELY-BELOW	- every point of X is below Y
DIRECTLY-BELOW	- COMPLETELY-BELOW & (WITHIN (X-SPAN X) (X-SPAN Y))
CENTERED-BELOW	- COMPLETELY-BELOW & (= (XCOR (CG X)) (XCOR (CG Y)))

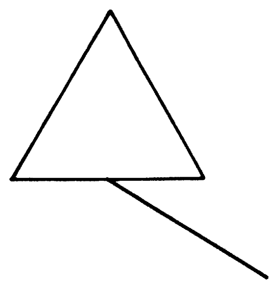
These four definitions are illustrated in figure B.2. "CG" returns the coordinates of the center of gravity of a vector structure. ABOVE, RIGHT-OF and LEFT-OF have similar definitions.

The user can specify the exact meaning. Alternatively, the system

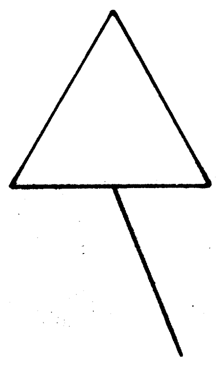
MEANINGS OF BELOW



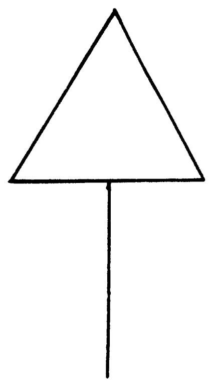
(PARTLY - BELOW TRUNK TOP)



(COMPLETELY-BELOW TRUNK TOP)



(DIRECTLY-BELOW TRUNK TOP)



(CENTERED-BELOW TRUNK TOP)

FIGURE B.2

will assume by default the usual meaning of COMPLETELY-BELOW. If this is found implausible due to the violations it implies, the system relaxes the constraint and tries less specific meanings. To minimize this search, the user can supply advice in the form of which meaning was intended.

M6 CONNECTED TOP TRUNK

There are several flavors of connectivity. Let P be the point of connection. Then (CONNECTED X Y) can assume the following meanings.

ENDPOINTS	- P is an endpoint of both a vector in X and a vector in Y.
TEE	- P is an endpoint of a vector in X and an interior point of a vector in Y.
INTERIORS	- P is an interior point of both a vector in X and a vector in Y.
OVERLAP	- X overlaps Y, i.e. X and Y share a sub-segment.

These four definitions are illustrated in figure B.3.

The user can specify the type of connection by inserting in the connectivity assertion:

(VIA <ENDPOINTS, TEE, INTERIORS, OVERLAP>).

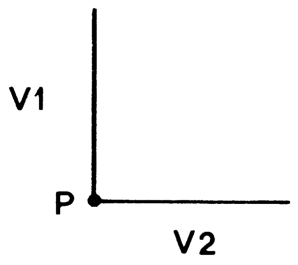
The default is that the connection involve at least one labeled vertex, i.e. ENDPOINTS or TEE.

The user can supply further specification by describing the connection point. For example, the connection between the TRUNK and the TOP of the TREE can be stated explicitly by:

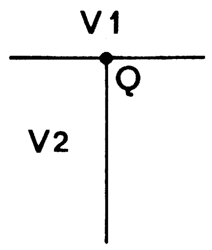
```
M6 (CONNECTED TOP TRUNK (AT P))
        WHERE M6.A (= P (ENDPOINT TRUNK))
              M6.B (= P (MIDDLE (SIDE TOP)))
```

ENDPOINT acquires meaning through the descriptive properties associated with vectors by the annotator. MIDDLE is simply a Cartesian function for computing the midpoint. SIDE TOP is defined by interpreting TOP in terms

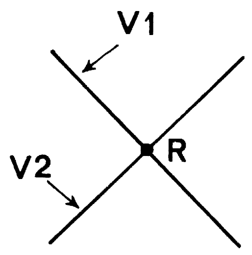
MEANINGS OF CONNECTED



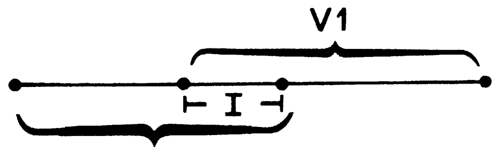
(CONNECTED V1 V2
(VIA ENDPOINTS)(AT P))



(CONNECTED V1 V2
(VIA TEE)(AT Q))



(CONNECTED V1 V2
(VIA INTERIORS)(AT R))



(CONNECTED V1 V2
(VIA OVERLAP)(AT I))

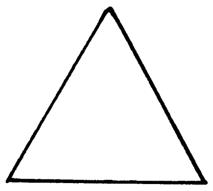
FIGURE B.3

of the TRIANGLE model.

B.2 TRIANGLE MODEL

An equilateral triangle is naturally described in terms of "typical elements" [Winston 1970] for the sides and rotations. Typical elements are useful when there are repeated occurrences of sub-parts having basically the same description.

```
MODEL TRIANGLE
M1 PARTS (SIDE 3) (ROTATION 3)
M2 FOR-EACH SIDE (= (LENGTH SIDE) 100)
M3 FOR-EACH ROTATION (= (DEGREES ROTATION) 120)
M4 RING CONNECTED SIDE
END
```



TRIANGLE

FIGURE B.4

M1 (PARTS (SIDE 3) (ROTATION 3)

The PARTS statement defines the generic names for the triangle's sides and rotations. The index for each equals the number of expected instances. A typical bug is for the control structure of an iterative or recursive program to result in the wrong number of instances.

M2 (FOR-EACH SIDE (= (LENGTH SIDE) 100))

M3 (FOR-EACH ROTATION (= (DEGREES ROTATION) 120)

The generic name SIDE refers to three vectors. The FOR-EACH requires that each of these vectors have length 100. Similarly, M3 constrains the DEGREES property of the three rotations to be 120.

It may be necessary to refer to the sides explicitly. This is obviously necessary when a generic model is associated with an explicit procedure. To accomplish this, the instances are numbered in the temporal sequence in which they occurred. For example, if the sides of the triangle are described by (PARTS (SIDE 3)), then the names (side 1), (side 2), and (side 3) are bound to each "FORWARD 100" in the following explicit program.

```
TO TRI
10 FORWARD 100          <- (accomplish (side 1))
20 RIGHT 120
30 FORWARD 100          <- (accomplish (side 2))
40 RIGHT 120
50 FORWARD 100          <- (accomplish (side 3))
60 RIGHT 120
END
```

M4 RING CONNECTED SIDE

"RING" applies the predicate CONNECTED to pairs of sides. The outcome is true if and only if there exists an ordering of the sides, S1 S2 S3 such that:

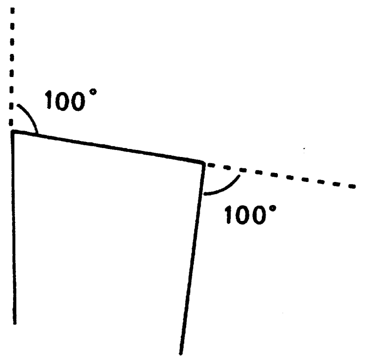
(CONNECTED S1 S2) & (CONNECTED S2 S3) & (CONNECTED S3 S1).

The default ordering is the temporal sequence in which the instances were generated. However, advice is possible referring to alternative orderings, e.g.

```
(RING CONNECTED SIDES (ANY ORDERING))
(RING CONNECTED SIDES (TEMPORAL ORDERING))
(RING CONNECTED SIDES (REORDER (TEMPORAL ORDERING) (2 1 3)))
```

The following illustrates an interpretation of the TRIANGLE model with respect to a program with bugs. The planning commentary has been generated by the Plan-Finder.

```
TO TRIANGLE1     <- (accomplish triangle)
10 FORWARD 100   <- (accomplish (side 1))
20 RIGHT 100     <- (accomplish (rotation 1))
20 FORWARD 100   <- (accomplish (side 2))
40 RIGHT 100     <- (accomplish (rotation 2))
50 FORWARD 100   <- (accomplish (side 3))
END
```



TRIANGLE 1

100° Rotations

FIGURE B.5

```
(INTERPRET (PROGRAM TRIANGLE1) (MODEL TRIANGLE))

        (INTERNAL VIOLATIONS IN MODEL PARTS)
                (NOT (= (ROTATION 1) 120))
                (NOT (= (ROTATION 2) 120))
                (NOT (EXISTS (ROTATION 3)))

        (INTERNAL VIOLATIONS BETWEEN MODEL PARTS)
                (NOT (CONNECTED (SIDE 3) (SIDE 1)))
```

Typical elements are particularly useful for describing the sub-picture produced by a round of an iterative or recursive program. The fact that the sub-picture is produced by the same segment of code naturally leads to similar descriptions for each occurence. An example is the following iterative triangle program:

```
TO ITERATIVE.TRIANGLE
10 MAKE "SIDES" 3
20 IF :SIDES=0 THEN STOP
30 FORWARD 100
40 RIGHT 120
50 MAKE "SIDES" :SIDES-1
60 GO 20
END
```

The typical elements SIDE and ROTATION mentioned in the model correspond to lines 30 and 40 of the program.

B.3 UNDERDETERMINED MODELS

A problem faced by program writing more than by debugging and plan-finding is that of the underdetermined model. Consider the FACE model.

```
MODEL FACE
M1 PARTS LEFT.EYE RIGHT.EYE NOSE MOUTH HEAD
M2 CIRCLE (HEAD LEFT.EYE RIGHT.EYE)
M3 TRIANGLE NOSE
M4 LINE MOUTH
M5 INSIDE (LEFT.EYE RIGHT.EYE NOSE MOUTH) HEAD
M6 ABOVE (LEFT.EYE RIGHT.EYE) NOSE
M7 BELOW MOUTH NOSE
END
```

This model simply does not state any special relationship between the eyes beyond that they are inside the head and above the nose. It does not state that they are of the same size, at the same height above the nose, and equally placed about a vertical line of symmetry extending through the center of the face. Figure B.6 illustrates unintended faces permitted by this description. Hence, it is impossible to determine the program fully.

Debugging and plan-finding escape this problem by having the user's program at their disposal. The program is presumably not far from correct. Hence, it implicitly provides these unspoken constraints. For program writing, however, there is no user-supplied code. Somehow, the additional

UNINTENDED FACES

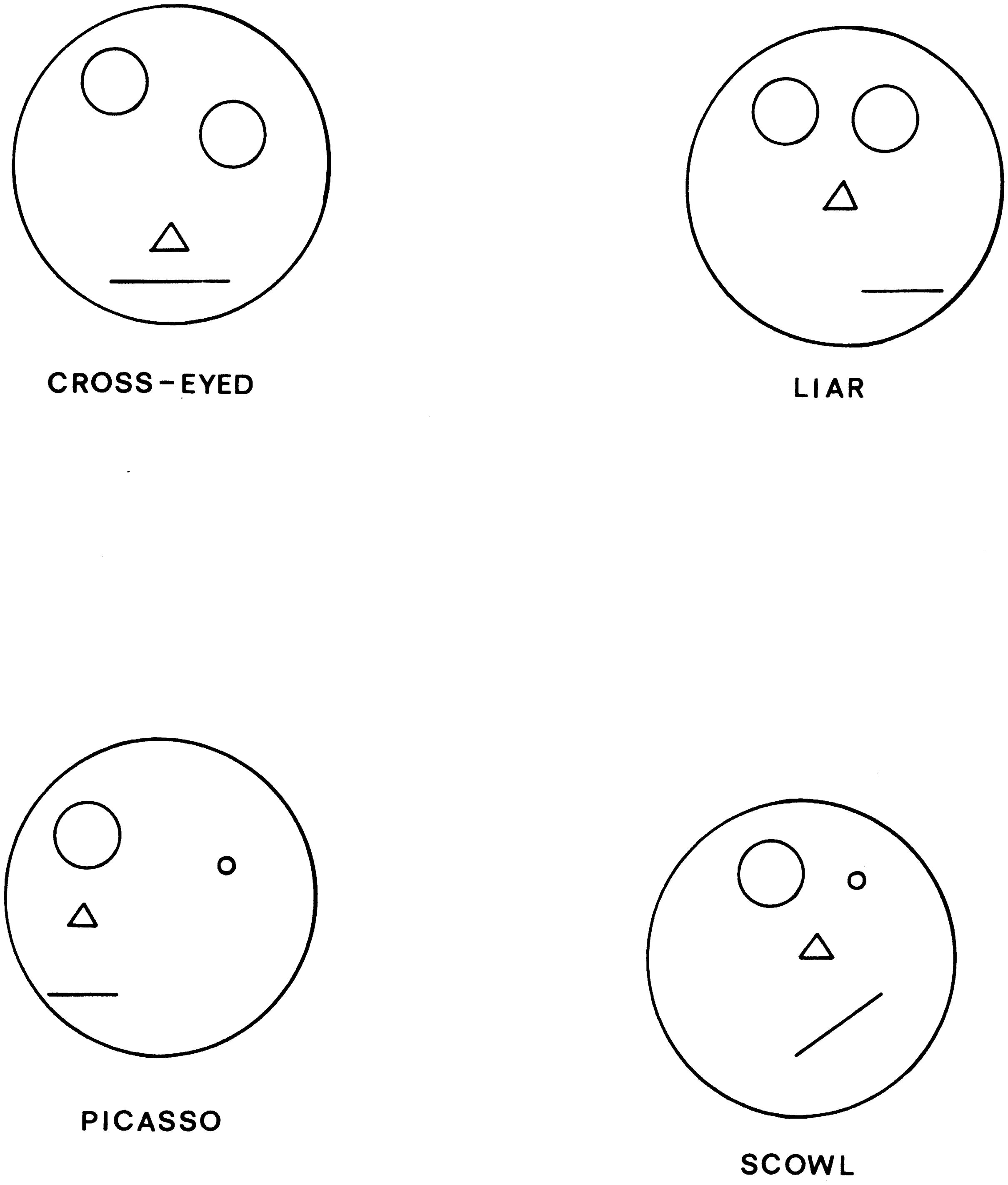


FIGURE B.6

model description must be provided.

One way is for the user to provide additional model statements.

```
(= (YCOORD LEFT.EYE) (YCOORD RIGHT.EYE))
(= (RADIUS LEFT.EYE) (RADIUS RIGHT.EYE))
(= (DISTANCE (CENTER NOSE) (CENTER LEFT.EYE))
   (DISTANCE (CENTER NOSE) (CENTER RIGHT.EYE)))
```

Alternatively, a sketch could be supplied. The sketch could fill in the numerical details for setting up the parts, determining inputs, and constraining relative size. Indeed, this naturally leads to a very interesting extension - that of generalizing an overdetermined sketch into a symbolic model.

Finally, the system could be provided with an epistemology for simple two-dimensional pictures such as faces, stickmen, wishing wells and trees to guide in supplying the absent detail.

B.4 DISAMBIGUATING MULTIPLE REFERENCE

Further elaboration of the model may be needed by the user when the original model accepts an unintended picture or in response to system queries directed at disambiguating the meaning of a model predicate or of a choice possibility. As indicated in the discussion of CONNECTED and BELOW for the TREE model, advice can be given as to the intended meaning and referenced objects. For example, the user can specify the intended location of the connection point.

```
M6 (CONNECTED TOP TRUNK (AT (AND (TRUNK ENDPOINT)
                                 (MIDDLE (TOP SIDE)))))
```

However, even with this advice for CONNECTED, more than one point is suggested. (ENDPOINT TRUNK) returns two possibilities: (SIDE TOP) returns three. In such cases, the predicate makes the most plausible choice from among the possibilities. The constraint is satisfied if any possibility wins. Thus, to establish the right binding for a bugged program with insufficient commentary, the ability to backtrack in a multiple choice situation is required. [Hewitt 1972, Sussman 1971]. This is considered in the "Plausible Search" procedures introduced in the plan-finding and debugging chapters.

B.5 LOGIC

Logical connectives such as AND, OR and NOT are manageable in models. They are rarely needed since the user has a specific picture in mind when he describes the intent of his program. However, the teacher may want to define a model for a set of pictures to aid in the debugging of a class project. This leads to the use of disjunctions.

"AND" is implicit in models as the set of statements is treated as a conjunction. However, for "NOT" statements, it is difficult to assign local responsibility to a part of the program for the negation. The negation is often accomplished implicitly.

> A deduction system would be useful here. (NOT (BELOW X Y)) may appear in the plan as attempting to achieve (ABOVE X Y). The imperative meaning of a negation is to achieve a particular positive predicate. Deductions for converting negations into positive model statements would aid in discovering the plan.

"OR" statements are a mechanism for defining a class of models, one for each of the disjuncts. Discovering the intended disjunct can be impossible where there are bugs. The simplest solution is to ask for advice.

B.6 MODELS FOR MORE COMPLEX PROGRAMS

For describing the picture drawn by a Fixed-Instruction Turtle Program, it is sufficient to use only primitive model predicates and sub-models. If more complex programs are allowed, then the descriptive power of a model must be increased. For example, if inputs and rounds are allowed, then it is possible to transform the triangle program into a procedure that draws any regular polygon.

```
TO POLY :SIDE :ANGLE
10 FORWARD :SIDE
20 RIGHT :ANGLE
30 POLY :SIDE :ANGLE
END
```

A variety of pictures drawn by this program are illustrated in Figure B.7.

A model is required that describes the class of pictures drawn by this single program. A natural generalization to the picture language to permit this is the use of variables.

```
MODEL POLY
M1 PARTS (SIDE ?N) (ANGLE ?N)
        ;?N is a variable which is instantiated by the
        ;interpretation process. The value is found by
        ;observing the number of SIDES in the picture.
        ;Naturally, this implies that the bug of "wrong number
        ;of sides" cannot be detected with this model.
M2 (RING CONNECTED SIDE)
        ;The default ordering for the ring of connections is
        ;the temporal order in which the sides are drawn.
M3 (RING EQUAL SIDE)
M4 (FOR-EACH ANGLE (= (DEGREES ANGLE) (QUOTIENT 360 ?N)))
END
```

Mathmematical facts can be inserted in the model: the relation between the external angle and number of sides of a polygon is effectively stated in the last line of the model.

For more complex programs, the model might itself be a recursive description and perhaps be executed in parallel with the program to establish the proper binding of code to parts. In this case, the parts list would behave very much like a local variable list with pushing and

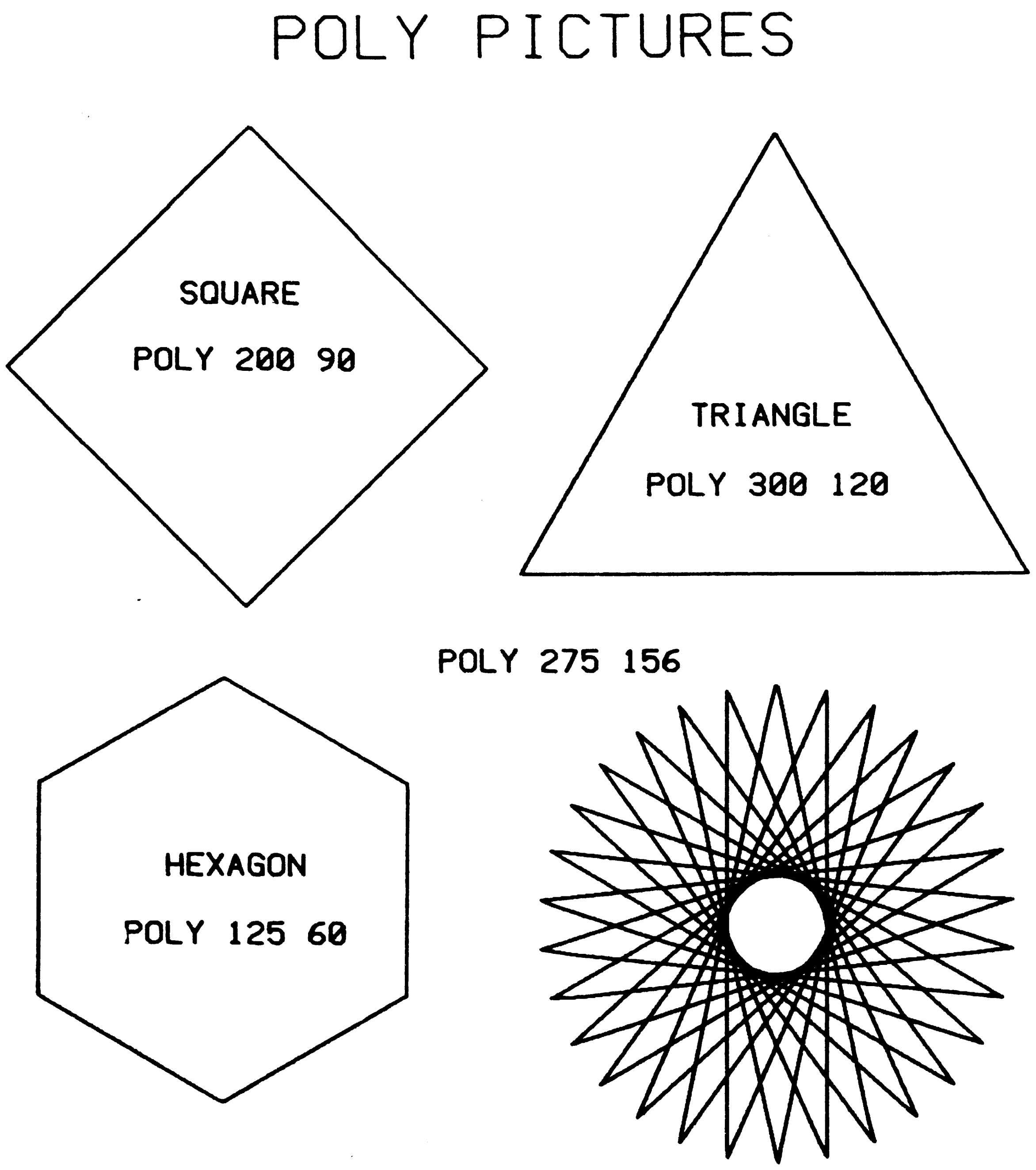

POLY PICTURES
SQUARE
POLY 200 90
TRIANGLE
POLY 300 120
POLY 275 156
HEXAGON
POLY 125 60

FIGURE B.7

popping of the binding.

B.7 MODELS AS NETS

Predicate models can be represented as nets (section 8.3). Using a uniform net-matching algorithm for "interpreting" the program's picture in terms of the model, however, is unsatisfactory. Bugs can make determining the proper match between model parts and sub-pictures very difficult. The net provides no guidance for the order in which the match is attempted. It is not able to draw on the various clues arising from the process, the program, and the form of the model. The plan-finding analysis, on the other hand, introduces a Plausible Search mechanism that is more intelligent about the binding of program to model.

For more complex pictures, models must be extended to allow variables (as in POLY), conditionals, and recursion. This type of extension goes beyond the net representation of pictures. It is required to handle the complexities caused by

interdependence of relations
dependence on context
abstract properties.

For example, in describing faces, there is a dependence between the allowable eyes and the allowable noses. For certain types of eyes, certain types of noses are required. For other types of eyes, noses are optional.

It is possible for a given relation to be sometimes necessary, other times not, depending on complex properties of the figure. This is not succinctly represented in nets. Complex dependencies can be handled by procedures through the use of conditionals. The second way in which nets are inadequate is in describing abstract properties. These may be more subtle than the "typical element". For example, the property of having "eyes" is a property of the scene. Eyes are usually two similar shapes inside a head. However, they can be dissimilar to show certain expressions. In other cases, they may overlap the head to get comic effects. The recognition program must be able to interrogate the context (net) describing the basic parts of the picture. But it must also be able to investigate the supercontext for special reasons that cartoon eyes or particular expressions might be used. The eye description is too complex--too many if's depending on other parts and maybe's depending on context--to be representable as a net.

APPENDIX C -- DEBUGGING A WISHING WELL PROCEDURE

This appendix presents a trace of the monitor system's performance in repairing a wishing well program including annotation of the user's program, finding the plan and debugging each violation. Figure C.1 shows the intended picture. The program is open-coded, rather than subroutinized, with the result that finding the proper plan and correction the bugs is correspondingly more complex. The program draws figure C.2 and is given below.

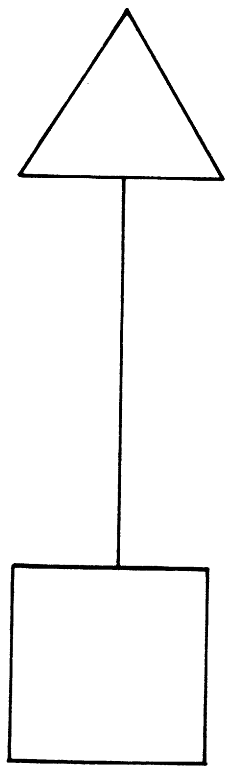

Intended WISHINGWELL

FIGURE C.1

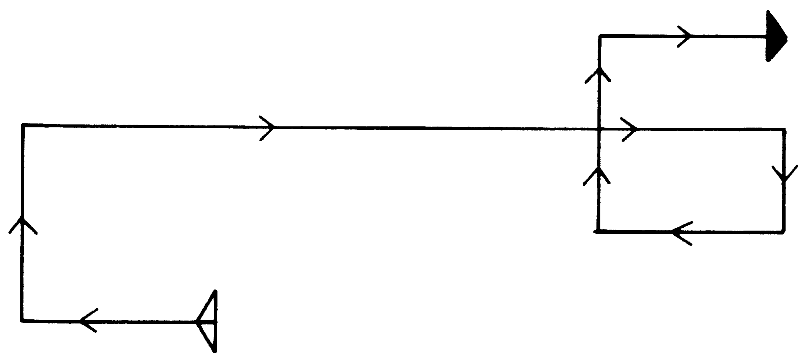


FIGURE C.2

```
TO WW            ;version 1
10 FD 100        ;Statements 10-50 are intended to accomplish the roof.
20 RT 90         ;This rotation is a bug: it should be 120 degrees.
30 FD 100
40 RT 90         ;This is the same bug as statement 20. The triangle
                 ;should be built from 120, not 90 degree rotations.
50 FD 100        ;The final side of the roof. It should be followed
                 ;by a preparatory step to move the turtle to the
                 ;proper initial position for drawing the pole. This
                 ;step is missing. This is reflected in the Rational
                 ;Form Violation of two contiguous FD instructions.
60 FD 200        ;This statement is intended to be the pole connecting
                 ;the roof to the well.
70 RT 90
80 FD 50         ;Statements 80-140 are intended to accomplish the well.
                 ;Statement 80 draws one half of the first side. This
                 ;is to cause the connection point between the pole
                 ;and the well to occur in the middle of the side.
90 RT 90
100 FD 100
110 RT 90
120 FD 100
130 RT 90
140 FD 100       ;The procedure should conclude with a final RT 90
                 ;and FD 50 to finish the side begun in statement 80.
                 ;However, this code is missing with the result
                 ;that the well is not a closed figure.
END
```

C.1 PERFORMANCE ANNOTATION

Even without the model, the system can annotate the performance of the program. For the graphic world of turtles, the annotation consists of describing the points, vectors and angles in the picture, recording relationships such as connectivity which are obvious from the turtle's local behavior, and generating suggestions for the Plan-finder and Debugger. (See chapter 6.) Figure C.3 shows the vectors of the picture named in the order in which they were drawn by the turtle. The local connectivity relations between sequential vectors is recorded in the database although the global connection between the interiors of vectors V4 and V7 is not noticed by the Annotator. Too much computation is required to test for all such connections in the absence of any motive from the model or plan.

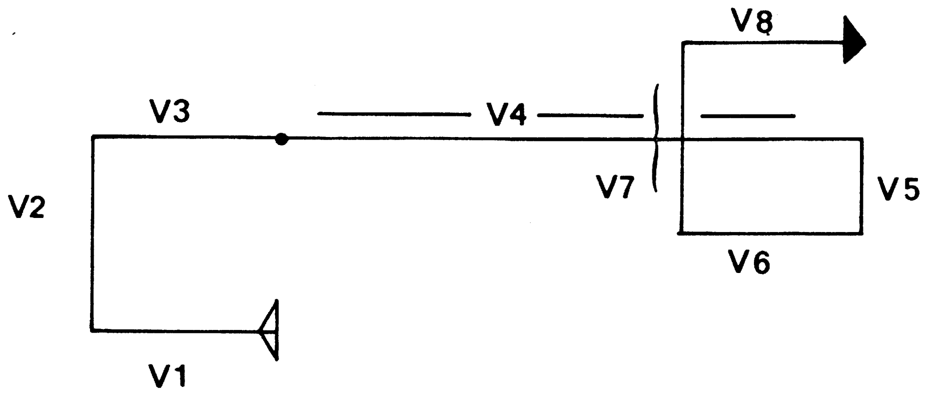


FIGURE C.3

The lengths of the vectors and the local connectivity at rotation points are not dependent upon the turtle's initial state and are therefore recorded as schematic properties. The orientation of the vectors are affected by the initial state and are part of the process annotation. The annotator does not bother to record ABOVE, RIGHT-OF, LEFT-OF, or BELOW relations because of the extent to which they depend upon the initial state.

No segmentation clues are available due to the open-coding of the program. However, statements 50 and 60, being two sequential FORWARD instructions, are a violation of rational form. This is recorded by a Caveat comment designed to lend credence to any repair hypotheses which commit the system to inserting rotations between these two statements.

C.2 THE WISHING WELL MODEL

For the monitor to continue the debugging process, the intended result must be described by the user. The following picture model describes the wishing well. The connection points between the roof, pole and well are constrained with some precision as otherwise the system would be unaware of certain bugs.

```
MODEL WISHINGWELL
M1 PARTS ROOF POLE WELL
M2 TRIANGLE ROOF
M3 LINE POLE
M4 SQUARE WELL
M5 ABOVE ROOF POLE WELL
M6 CONNECTED WELL POLE (AT P)
        M7 (= P (MIDDLE (UPPER (SIDE WELL))))
        M8 (= P (BOTTOM (ENDPOINT POLE)))
M9 CONNECTED POLE ROOF (AT Q)
        M10 (= Q (MIDDLE (BOTTOM (SIDE ROOF))))
        M11 (= Q ((UPPER (ENDPOINT POLE)))
M12 HORIZONTAL (BOTTOM (SIDE ROOF))
M13 HORIZONTAL (UPPER (SIDE WELL))
END
```

This model uses as sub-models descriptions for a square and an equilateral triangle. These two sub-models are quite similar and hence only the triangle model is given below.

```
MODEL TRIANGLE  ;generic model
M1 PARTS (SIDE 3) (ROTATION 3)
M2 FOR-EACH SIDE (= (LENGTH SIDE) 100)
M3 FOR-EACH ROTATION (= (DEGREES ROTATION) 120)
M4 RING CONNECTED SIDE
END
```

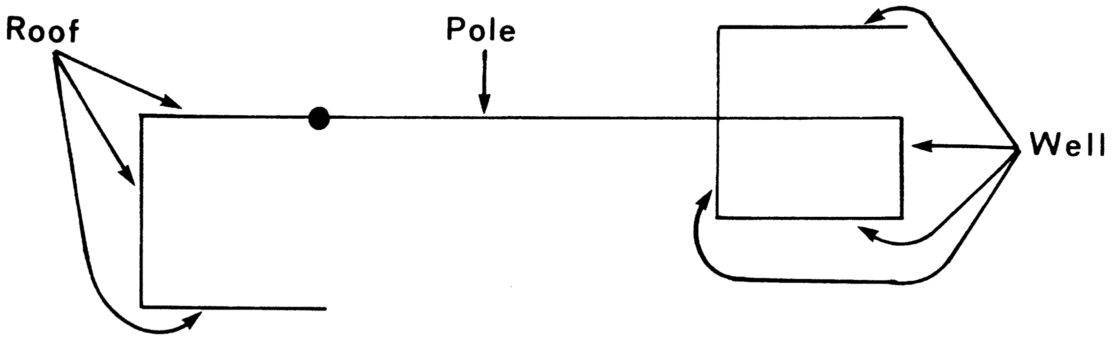
See section B.2 for a discussion of generic models.

C.3 FINDING THE PLAN

To understand the relation of the program and its picture to the model, the plan must be found. No clues are available in the form of global connections at endpoints or sub-procedures. Therefore, the system begins a search of the space of partial plans beginning with an analysis of the first statement of the program.

Statement 10 draws V1. There are three possible interpretations: V1 is a part of either the TOP, the POLE or the WELL. However, program-writing criteria related to accomplishing transitive predicates apply. In particular, the description (ABOVE ROOF POLE WELL) suggests to the plan-finder that the parts are accomplished in transitive order; hence, the partial plan in which statement 10 is interpreted as part of the POLE is hung, i.e. held in abeyance until the more fruitful partial plans either prove complete or are themselves found too implausible. (In the latter case, the hung plans are once again investigated). This leaves two possible linear linear plans:

```
PLAN1    ;roof -> pole -> well
PLAN2    ;well -> pole -> roof.
```

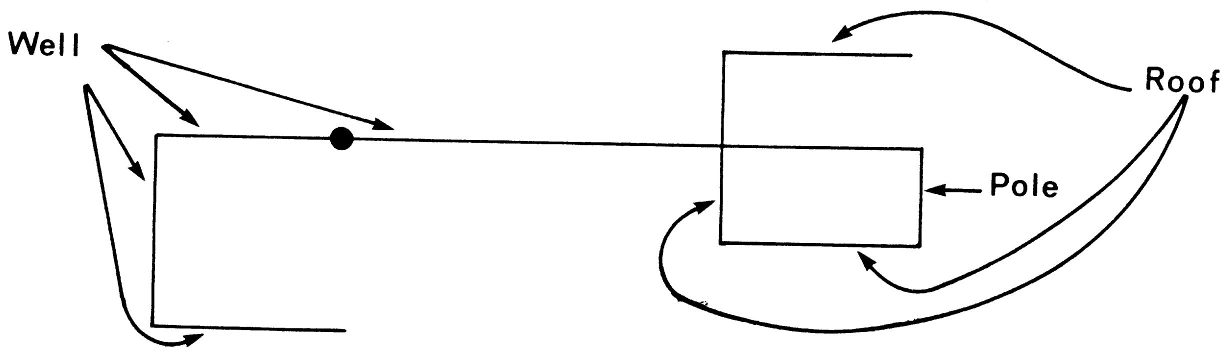


PLAN1

FIGURE C.4

PLAN1 (figure C.4) interprets statements 10 through 50 (vectors V1, V2 and V3) as the ROOF. Statement 60 is considered to be the POLE and the remainder of the procedure to be the WELL. Under this binding of the picture to the model, the following model statements are violated.

```
(INTERPRET (PROGRAM WW) (MODEL WISHINGWELL) (PLAN PLAN1))

;Violations of Properties of Parts of WW
       (NOT (TRIANGLE (WW 10-50)))
              (NOT (= (DEGREES (ROTATION 1)) 120))
              (NOT (= (DEGREES (ROTATION 2)) 120))
              (NOT (= (DEGREES (ROTATION 3)) 120))

       (NOT (SQUARE (WW 80-140)))
              (NOT (EQUAL SIDES))
              (NOT (CLOSED SIDES))

;Violations of Relations between Parts of WW
  ;Intrinsic Violations
       (NOT (= Q (MIDDLE (SIDE ROOF))))
       (NOT (= P (MIDDLE (SIDE WELL))))

 ;Extrinsic Violations
       (NOT (ABOVE ROOF POLE))
       (NOT (ABOVE POLE WELL))
```



PLAN2

FIGURE C.5

PLAN2 (figure C.5) analyzes the program in the opposite way: the first four vectors are interpreted as the WELL, statement 80 (vector V5) as the POLE and the remainder of the program (vectors V6, V7 and V8) as the ROOF. Surprisingly, despite this different binding of the picture to the model, the same set of violations is produced with the addition that the third rotation of the square between V3 and V4 is missing, i.e. (NOT (EXISTS (ROTATION 3 SQUARE))).

PLAN1 implies one less violation than PLAN2 and is consequently passed to the debugger. The user can alter this choice by explicitly indicating his preference. This is one of those situations where multiple bugs prevent any clear selection of the intended plan. If WW had been correct, the global connections at P between the WELL and the POLE and at Q between the POLE and the ROOF would have suggested the segmentation of the program into main-steps. It is the bugs combined with the open-coding that makes proper interpretation difficult.

The pretty-print based on PLAN1 clearly reveals the structure of the program. For readability, the code for the triangle and square is named TRI and SQ: the system would not generate such mnemonic names although it would treat the open-coded sequences for model parts as sub-procedures.

```
TO WW3          <- (accomplish wishingwell)
10 TRI          <- (open-coded sequence for roof)
60 FD 200       <- (accomplish pole)
70 RT 90        <- (setup heading)
80 SQ           <- (open-coded sequence for well)
END

TO TRI          <- (accomplish triangle)
10 FD 100       <- (accomplish (side 1 triangle))
20 RT 90        <- (accomplish (rotation 1 triangle))
30 FD 100       <- (accomplish (side 2 triangle))
40 RT 90        <- (accomplish (rotation 2 triangle))
50 FD 100       <- (accomplish (side 3 triangle))
END

TO SQ           <- (accomplish square)
80 FD 50        <- (accomplish (side 1 square))
90 LT 90        <- (accomplish (rotation 1 square))
100 FD 100      <- (accomplish (side 2 square))
110 LT 90       <- (accomplish (rotation 2 square))
120 FD 100      <- (accomplish (side 3 square))
130 RT 90       <- (accomplish (rotation 3 square))
140 FD 100      <- (accomplish (side 4 square)
END
```

C.4 FIXING THE ROOF

The debugger first repairs violated properties. Hence, debugging begins by repairing the ROOF.

(FIX (TRIANGLE TRI))

The first debugging strategy is to runthe responsible code in private. Consequently, TRI is executed with the turtle initialized at the HOME state. The result is the open rectilinear chain of figure C.6. The program is commented with respect to the TRIANGLE model shown in section C.2.

```
TO TRI          ;extracted from WW, version 1
10 FD 100       <- (accomplish (side 1 triangle))
20 RT 90        <- (accomplish (rotation 1 triangle))
30 FD 100       <- (accomplish (side 2 triangle))
40 RT 90        <- (accomplish (rotation 2 triangle))
50 FD 100       <- (accomplish (side 3 triangle))
END
```

Given this binding of code to model parts, the violated model statements ordered by the usual criteria are:

```
;Violations of Properties of TRI
        (NOT (= (DEGREES (ROTATION 1)) 120)
        (NOT (= (DEGREES (ROTATION 2)) 120)
        (NOT (= (DEGREES (ROTATION 3)) 120)

;Violations of Relations between Parts of TRI
        (NOT (CONNECTED (SIDE 3) (SIDE 1)))
```

Debugging the rotations is made simple by the numeric description of the rotations supplied by the model. This illustrates, first of all, the fashion in which debugging becomes simpler as the commentary becomes more explicit. The imperative semantics for "=" suggests that "(P A)" be made equal to "(P B)". The semantics for DEGREES indicates that this

DEBUGGING THE ROOF OF WW3

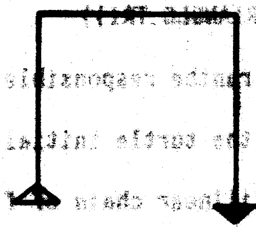


FIGURE C.6

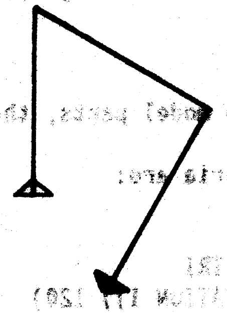


FIGURE C.7

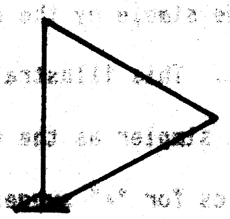


FIGURE C.8

change is accomplished by inserting an approriate rotation.

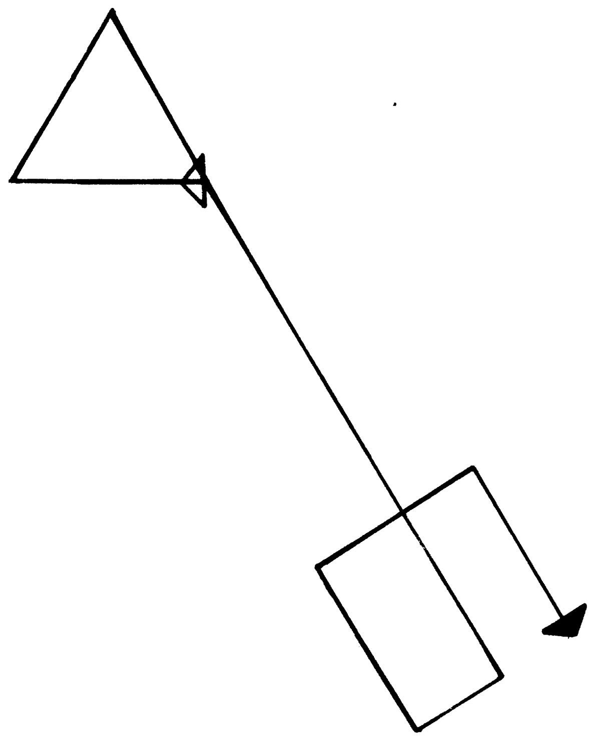
(= (DEGREES ROTATION) :A) => (= (INPUT RIGHT) :A)

(This is the same semantic statement used by the annotator but in the <= direction.) The insertion of RIGHT 120 is done for each rotation (see figures C.7 and C.8) and the result is the program given below.

```
TO TRI          ;WW version 3, satisfies visible model.
10 FD 100
20 RT 120       <- (= (degrees (rotation 1 triangle)) 120)
30 FD 100
40 RT 120       <- (= (degrees (rotation 2 triangle)) 120)
50 FD 100
END
```

The violation (NOT (CONNECTED (SIDE 3) (SIDE 1))) is corrected as a side effect of fixing the rotations of the triangle. Using the corrected TRI code, WW draws figure C.9.

An important issue in the theory of debugging is the extent to which making repairs depends upon specific domain-dependent knowledge versus general techniques of wide applicability. Suppose the model for TRIANGLE did not specify the rotations: could the procedure be debugged? The only violation would be the lack of connectivity. The answer is that there is no "General Debugging Knowledge" that would apply. Debugging would be dependent on knowing the theorem that a regular polygon will close if the rotations are equal to (number of sides)/360. The point is that debugging competence, especially where global bugs are concerned, depends on insight into the domain. This is to be expected. Simply knowing a language (in this case, that of procedures) cannot free the speaker of knowing what he is talking about. There are important facts to be known about planning, debugging and description; but these complement and cannot replace understanding the problem domain.



WW
Version 1

FIGURE C.9

TRI, version 3, is a typical non-transparent triangle procedure. The model, however, specified three rotations. The program satisfies the "visible" part of the model so this inconsistency is tolerable. However, it may be a bug (as opposed to over-specification in the model). Therefore, it is remembered as a Caveat comment. . This treatment is identical to that of oddities noticed by the annotator with respect to missing rotations. The comment is used only if subsequent debugging indicates that the possible cause of some violation is a missing rotation at the location where (ROTATION 3 ROOF) was expected. The caveat then adds plausibility to the strategy and contributes its understanding of the underlying cause.

If the debugger is considering a correction to the heading in the code immediately following TRIANGLE, then the caveat is observed and an edit is made to insert the missing rotation in TRIANGLE.

This caveat in fact does play a debugging role in fixing the connection between the pole and the roof. See section C.6.

C.5 REPAIRING THE WELL

(FIX (SQUARE SQ))

Correcting the WELL to be a SQUARE is the second main-step that needs fixing. The SQUARE model is similar to the TRIANGLE model except for the information that there are four sides and that the rotation is 90 degrees. On the basis of PLAN1, the commentary for statements 80-140 is shown below.

```
TO SQ          ;extracted from WW version 1, see figure C.10.
80 FD 50       <- (accomplish (side 1 square))
90 RT 90       <- (accomplish (rotation 1 square))
100 FD 100     <- (accomplish (side 2 square))
110 RT 90      <- (accomplish (rotation 2 square))
120 FD 100     <- (accomplish (side 3 square))
130 RT 90      <- (accomplish (rotation 4 square))
140 FD 100     <- (accomplish (side 4 square))
END
```

(FIX (= (SIDE 1 WELL) 100))

This violation is more difficult to debug than the preceding problem with the ROOF. The square is missing a part of the first side (See figure C.10.). The debugger has several options:

1. The first is to modify the existing code.

2. The second is to find some part of the picture (i.e. program) which may serve the additional purpose, beyond the one assigned to it in the plan, of being the completion of side 1 of the square.

3. The third is to reject the plan and call for a new one from the plan-finder which does not explain side 1 of the triangle as statement 80 of WW.

The first of these possibilities is examined. Only if there is no plausible way to insert the code are the other debugging routes pursued.

There are two ways to modify the SQ code. The first is to alter the input to some existing FD instruction: the second and more drastic way is to add new code. This produces the square in figure C.11 and the

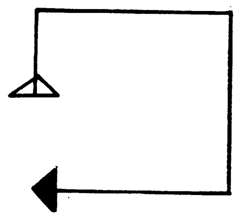
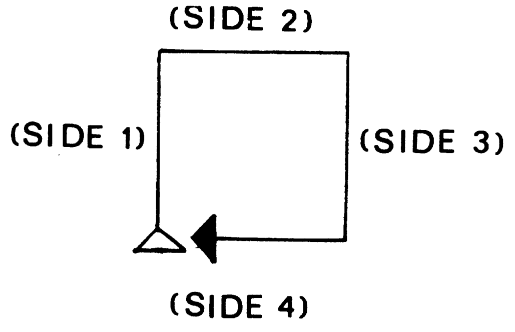


FIGURE C.10



SIDE 1 LENGTHENED

FIGURE C.11

corresponding wishing well picture of figure C.12. It remains only to correct the location of the connection points between the WELL, POLE and ROOF. The resulting editing strategy is sufficient to fix all of the violations.

The user, however, probably intended statement 80 to be only FD 50. His plan was to have the starting position of the SQ be the proper connection point to the POLE. This cannot be known while debugging the SQ code in private. Private Debugging deliberately ignores the relationships between the bugged object and other model parts in order to simplify finding the repair.

To explore the elementary code-writing capabilities of the

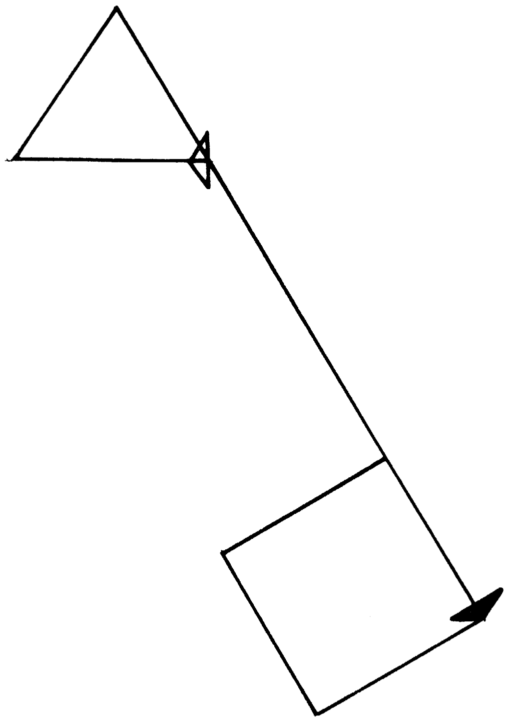


FIGURE C.12

debugger, let us assume that the user has prohibited modification of the SQ code by protecting statements 80 to 140.

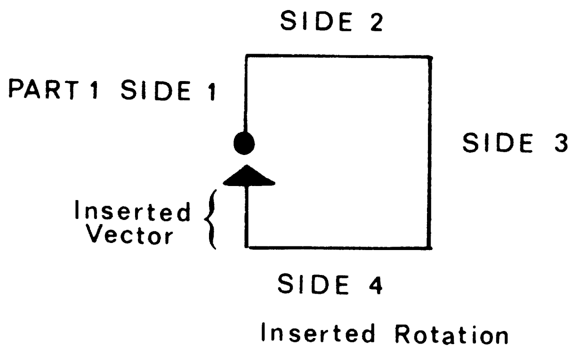
```
(PROTECT (STATEMENTS 80-100) USER-ADVICE)
```

In such an event, the debugger attempts to complete the short side by inserting code for a vector. The code-writer for vectors is based on the following imperative semantics: a vector is accomplished by determining either (1) the starting point, direction and length or (2) the endpoints.

To determine the necessary information, the model constraints on (side 1) are examined.

```
(CONNECTED (SIDE 1) (SIDE 4))
(CONNECTED (SIDE 1) (SIDE 2))
(= (LENGTH (SIDE 1)) 100)
```

The connectivity predicates indicate both of the desired endpoints. Therefore, an explicit request can be passed to the state editor to create the needed vector. The result is the following SQUARE program.



CORRECTED SQUARE

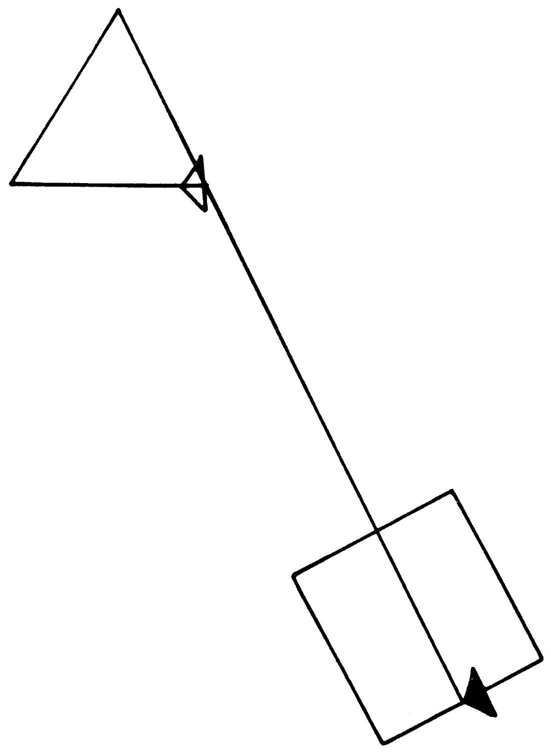
FIGURE C.13

```
TO SQ          ;WW version 4, satisfies visible model for square
80 FD 50       <- (accomplish (piece 1 (side 1 square)))
90 RT 90
100 FD 100
110 RT 90
120 FD 100
130 RT 90
140 FD 100
150 RT 90      <- (addcode (rotation 4 square) such-that
                          (= (degrees (rotation 4 square)) 90))
160 FD 50      <- (addcode (piece 2 (side 1 square)) such-that
                          (= (length (side 1 square)) 100))
END
```

Figure C.14 is the wishing well drawn by using the corrected code for the roof and well.

C.6 FIXING THE TOPOLOGY OF THE WISHING WELL

The parts of the wishing well are now successfully accomplished. The remainder of this debugging scenario treats SQ and TRI as inviolate code sequences just as though they had been originally written as subprocedures by the user. The identification of common purpose reduces the complexity of analyzing the open-coded program. The next debugging



WW
Version 4

FIGURE C.14

task is to insure that the parts are in the proper relation with one another. The following intrinsic (i.e. topological) violations exist:

```
(NOT (CONNECTED POLE ROOF (AT (MIDDLE (BOTTOM (SIDE ROOF))))))
(NOT (OUTSIDE POLE WELL))
      ;This constraint is not explicit in the model but is
      ;a necessary condition for (COMPLETELY-ABOVE POLE WELL).
```

```
(FIX (CONNECTED POLE ROOF (AT (MIDDLE (BOTTOM (SIDE ROOF))))))
```

Connections are initially analyzed as Local Preparation Errors and possible culpable interfaces are confined to those between the main-steps for the input parts. Connectivity is debugged by finding or hypothesizing the connection point, computing the required translation and then inserting it into the most likely interface.

The connection point Q is described by:

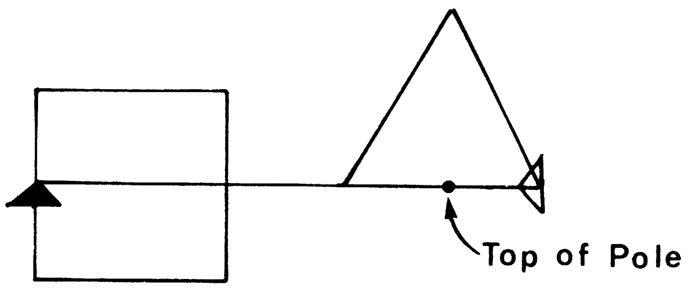
```
(= Q (MIDDLE (BOTTOM (SIDE ROOF))))
```

"SIDE ROOF" can refer to any of the three sides: branching occurs on the basis of which is chosen. Preference is given to the currently "bottom" side. If this debugging strategy is rejected, the search considers alternative choices for the intended (BOTTOM SIDE).

Under the direction of the CONNECTION semantics, the system computes the vector V (direction 270 degrees, length 50) necessary to translate the POLE such that the connection is at the midpoint of the bottom side. This translation is not inserted directly into the TRI procedure: main-steps are inviolate in linear debugging. The only remaining editing locus between main-steps is at the interface between the ROOF (statements 10-50) and the POLE (statement 60).

```
(STATE.EDIT (BETWEEN (CODE POLE) (CODE ROOF)) (INSERT V))
```

The resulting edit is the insertion of "RT 120" followed by "FD 50" which achieves the desired effect. This produces WW version 5 and draws figure C.15.



WW
Version 5

FIGURE C.15

```
TO WW           ;WW version 5, (connected pole roof) fixed.
10 TRI          <- (open-coded sequence for roof)
53 RT 120       <- (setup heading such-that
                    (retrace (interface statement 55) (side 3 roof)))
55 FD 50        <- (retrace (side 3 roof) such-that
                    (= Q (middle (side roof))))
60 FD 200       <- (accomplish pole)
70 RT 90
80 SQ           <- (open-coded sequence for well)
END
```

This debugging edit is confirmed by the comment created as a result of the existence of only two rotations in the triangle. The inserted RT 120 constitutes a third. Thus, this insertion is preferred over the alternative of inserting RT 180 in order to retrace to the middle of side 2.

(FIX (OUTSIDE POLE WELL))

The following represents a trace of the Debugger in repairing this violation:

;debugging technique

1. Begin with a linear debugging attack. Hypothesize a Local Preparation Error. Search for the culpable interface.

2. OUTSIDE is classified as an intrinsic violation. Hence, the error is local. Therefore, restrict possible interfaces to those in causal chain between main-steps for input parts.

3. Hypothesize the most likely culpable interface to be immediately preceding the second main-step.

;imperative semantics

1. The semantics for OUTSIDE direct that if the initial position of the interface is on the border, a fix is to have the exit heading of the interface point "out" rather than "in". "Out" is defined as a heading that causes the sub-figure to be rotated as a rigid body such that

it is outside.

2. A range of headings satisfies this constraint for the POLE and the WELL: the average heading of 180 is selected by default.

(STATE.EDIT (BETWEEN (CODE POLE) (CODE WELL)) (= HEADING 180))

The State Editor, using Rational Form criteria, merges this fix into statement 70 with the result that statement 70 becomes "LEFT 90". WW version 6 is produced and draws figure C.16.

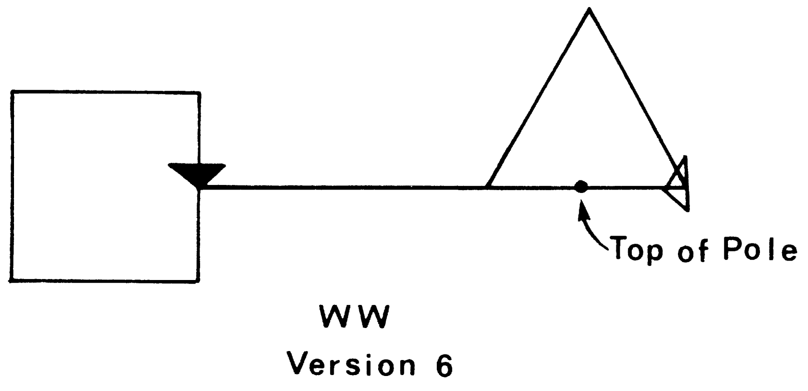


FIGURE C.16

```
TO WW            ;WW version 6, (outside pole well) fixed.
10 TRI           <- (open-coded sequence for roof)
53 RT 120        <- (setup heading such-that
                      (retrace (interface statement 55) (side 3 roof)))
55 FD 50         <- (retrace (side 3 roof) such-that
                      (= Q (middle (side roof))))
60 FD 200        <- (accomplish pole)
70 LT 90         <- (setup heading such-that (outside pole well))
80 SQ            <- (open-coded sequence for well)
END
```

C.7 DEBUGGING THE ABOVE RELATIONS

The remaining violationed relations are extrinsic, i.e. possibly caused by the frame of reference. These violations are:

(NOT (COMPLETELY-ABOVE ROOF POLE))

(NOT (COMPLETELY-ABOVE POLE WELL))

(FIX (ABOVE ROOF POLE))

Following a linear attack, the analysis is to find the culpable interface. For each interface, the semantics for ABOVE indicate the desired state change. Two possible interfaces are considered: the initial setup and the intermediate interface between the ROOF and the POLE. For the initial interface, a rotation of LEFT 90 (as statement 5) causes the ROOF to be above the POLE and, as a beneficial side effect, the POLE to be in turn above the WELL. However, this correction causes the bottom side of the ROOF to no longer be horizontal.

Alternatively, correcting the above relation by inserting LT 90 as statement 57 corrects the violation, has the same beneficial side effect and does not undo the orientation of the ROOF. Therefore this correction is chosen.

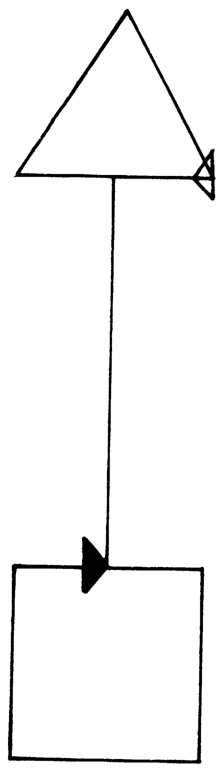
(STATE.EDIT (BETWEEN (CODE ROOF) (CODE POLE)) (= HEADING 180))

The resulting program with the POLE properly above the WELL is:

```
TO WW          ;WW version 7, (above roof pole) fixed.
10 TRI         <- (open-coded sequence for roof)
53 LT 120      <- (setup heading such-that
                   (retrace (interface statement 55) (side 3 roof)))
55 FD 50       <- (retrace (side 3 roof) such-that
                   (= Q (middle (side roof))))
57 LT 90       <- (setup heading such-that (above pole well))
               <- (assume (= (entry heading) 270))
               <- (protect heading through WW statement 60)
60 FD 200      <- (accomplish pole)
70 LT 90       <- (setup heading such-that (outside pole well))
80 SQ          <- (open-coded sequence for well)
END
```

(FIX (ABOVE ROOF WELL))

Fixing the interface between the ROOF and the POLE has the



WW
Version 7

FIGURE C.17

beneficial side effect of putting the WELL above the POLE. Consequently, the bug is fixed with the explanation that the same underlying cause of Local Preparation Error which accounted for "(NOT (ABOVE POLE WELL))" is responsible for this error as well. Protection of the "heading" is extended to statement 80.

```
TO WW          ;WW version 8, (above pole well) fixed.
10 TRI         <- (open-coded sequence for roof)
53 LT 120      <- (setup heading such-that
                   (retrace (interface statement 55) (side 3 roof)))
55 FD 50       <- (retrace (side 3 roof) such-that
                   (= Q (middle (side roof))))
57 LT 90       <- (setup heading such-that (above pole well))
               <- (assume (= (entry heading) 270))
               <- (protect heading through WW statement 80)
60 FD 200      <- (accomplish pole)
70 LT 90       <- (setup heading such-that (outside pole well))
80 SQ          <- (open-coded sequence for well)
END
```

The above program is the final result and the intended picture is achieve

APPENDIX D -- MORE ON IMPERATIVE SEMANTICS

The following discussion describes the imperative knowledge associated with the model primitives in greater detail. Recall that the purpose of the imperative semantics for the model primitives is to propose specific edits at a given point in the program to correct a violation of the primitive. As a convention, X and Y will represent sub-pictures, i.e. sets of vectors in the picture drawn by the program while (CODE X) and (CODE Y) are the code in the program which drew the sub-pictures.

D.1 GEOMETRIC PRIMITIVES

(LINE X) <=> (OR (VECTOR X) (AND (VECTORS X) (PARALLEL X) (CHAIN X)))

A picture is a line if and only if it is either a primitive vector or a sequence of collinear vectors. The imperative semantics for OR and AND direct the Debugger either to reduce a set of non-collinear vectors to a single vector or to make them parallel and connnected. Imperative semantics for PARALLEL and CONNECTED are provided below.

(VECTOR X)

A vector is the result of executing the turtle primitives FORWARD or BACK. If X is a set of vectors, then the repair is to delete all but one element of the set. This deletion is accomplished either by deleting the responsible code or by modifying the plan so that the interpretation does not commit all of the vectors to be part of X. See the plan-finding scenario for TREE1 (section 7.7) as an example of heterarchy between the Debugger and the Plan-finder.

(PARALLEL X) <=> (FOR-EACH A,B IN X (= (DIRECTION A) (DIRECTION B)))

X is a set of vectors. Each vector has a direction. The repair is to alter the direction of some subset of vectors so that all of the elements of X have the same direction. The imperative semantics of FOR-EACH guides the choice of the subset to be affected. Generally, the first vector is considered correct and the remainder are made parallel to it. The annotation semantics for vectors, section 6.3, provides the link between turtle primitives and the direction property of vectors.

```
(VERTICAL X) <=> (OR (= (DIRECTION X) 0) (= (DIRECTION X) 180))
```

If X is a vector, then alter preceding rotations so as to make its direction 0 or 180. If X is a set of vectors, then accomplish this for every element.

```
(HORIZONTAL X) <=> (OR (= (DIRECTION X) 90) (= (DIRECTION X) 270))

(INVISIBLE X) <=> (OR (= (PENSTATE X) :UP)
                      (EXISTS Y SUCH-THAT (RETRACE X Y)))
```

Interface vectors are supposed to be invisible. This can be accomplished in two ways. They can be drawn with the pen up or they can overlay some picture vector.

(RETRACE X Y)

To cause X to retrace Y, find a visible part Y. If possible, choose Y such-that it is already connected to X. If this is not possible, then connect an endpoint of X to Y. The next step is to cause X to overlap Y. If the connection is at an endpoint, then this is accomplished simply by altering the direction of X. X must end on Y so that it is invisible. If X is a chain of vectors, then each element of X must retrace some element of Y.

(CONNECTED X Y)

Assume that X is accomplished after Y. Choose a connection point on X, say P1, and a connection point on Y, say P2. If the exact position is unknown, deduce it from constraints such as preferring minimal changes to be made to the code. This has the imperative consequence of searching for the connection point on Y by manipulating individually the length and angle inputs to translation and rotation interface steps. Branch in considering alternative allowable connection positions. Debugging TREE1 so that the trunk properly connects to the top is an example. Then compute the vector V from P1 to P2. The edit is to insert code for V into an interface between X and Y. By the Rigid Body Theorem, this will have the effect of translating X so that P1 is moved to coincide with P2.

(ABOVE X Y) - (similar technique for BELOW, RIGHT-OF, LEFT-OF)

To compute the required correction for a given interface: assume that the figure has already been debugged to be topologically correct--e.g. all of the connections are correct. This implies that the only degree of freedom in interfaces is the heading.

In considering a given interface, find the range of headings which satisfy the predicate. The range is determined by first finding the heading of most restrictive meaning of ABOVE -- CENTERED-ABOVE wherein the center of gravity of X is directly above Y. Then relax this heading to find the maximum range in which less restrictive meanings of the predicate--COMPLETELY-ABOVE and PARTLY-ABOVE--remain true. To select a specific heading to actually insert into the code, choose the value that satisfies the most restrictive meaning of ABOVE. If there is still a range of possible headings, use the average value. Record the range considered in case later debugging results in conflicts and another heading must be chosen.

(INSIDE X Y) - (similar technique for OUTSIDE, OVERLAP)

INSIDE, OUTSIDE and OVERLAP do not depend on the global frame of reference; they are rotation and translation invariant. Hence, the guilty interface can be restricted to one occuring between X and Y. Interfaces prior to X (assuming that X is accomplished first) need not be considered. This is in contrast to predicates like ABOVE, BELOW, RIGHT-OF and LEFT-OF whose truth value is affected by rotation.

Linear corrections required at a given interface:

(Assume that Y is accomplished first.)

1. If the entry position to the interface is on the border, then insert a rotation such that the position change of the interface moves X into the interior of Y. If, as a result of this, X comes out the other side of Y, then decrease the length of the position change as well.

2. If the entry position of the interface is in Y, then decrease the length of the position change accomplished by the interface until X is inside Y.

3. If X and Y are achieved globally, then it will be very difficult to find a simple repair to some interface between X and Y. The turtle is not attempting some direct course from X to Y. The exact starting position of X may have to be found.

Non-linear corrections to main-steps for X and Y:

If (1) linear debugging is unsuccessful, i.e. no modification of any interface is possible that does not introduce more violations and (2) the size of the part was not specified in the model, then "scale-change" becomes a possible remedy. This is the technique used to correct the GOOGLY.EYES face (figure 3.11). Specifically, if X is overlapping Y, then either X can be shrunk or Y expanded. This is accomplished by changing the scale factor, i.e. multiplying the input to all vector instructions by the same constant. Altering the scale of X does not introduce any shape change and should therefore not cause any new violations of properties of X. The preferred change -- shrinking X or expanding Y -- is decided by minimizing bad side effects of the change.

D.2 LOGICAL PRIMITIVES

A. Equality

To fix (= (P A) (P B)), use the imperative semantics for property P to either make (P A) equal to (P B) or vice versa. Choose whether to alter A or B on the basis of the usual debugging plausibility criteria of avoiding conflicts, minimizing change to the user's program and preferring beneficial side effects.

B. Logical Connectives

To fix (OR D1 D2 ...), repair one disjunct. In the absence of special knowledge, set up parallel debugging strategies, each focused on a different disjunct. Estimate plausibility of the preferred disjunction on the basis of the estimated effort to make it come true. For example, a disjunct that is itself a conjunction can be judged by how many of the conjuncts are already true. An example is the meaning of LINE as "(AND (VECTORS X) (PARALLEL X) (CHAIN X))".

To fix (AND C1 C2 ...), correct all of the conjuncts. Order the debugging attack on the basis of dependency, i.e. correct conjuncts constraining main-steps before correcting predicates describing the relations between main-steps. A given group of violations at the same level are debugged in temporal order. These are the criteria that were used to order the initial set of violations.

To fix (NOT P), find the relation representing the negation of P, e.g. ABOVE - BELOW, RIGHT - LEFT, or INSIDE - OUTSIDE, and establish it. In general, the system really has no particular mechanism (or need) to make a given predicate untrue. It does not care what extra predicates are true of the picture so long as the model statements are correct. i.e. it is not operating under the assumption that only the model predicates should be true of the program.

C. Set Deletion

To make (SET X) into (SUBSET Y), eliminate (X - Y) by (1) deleting code or (2) explaining (X - Y) as not part of X by finding a new plan.

D. (FOR-EACH (X1, X2, ... IN X) (P1 & P2 & ...))

Debug those Xi which do not satisfy the predicates. Assume as correct the majority view. For example, if the goal is to make a set of vectors parallel, and a majority already are parallel and have some heading, then debug the direction of the minority to equal this heading.

APPENDIX E -- MORE ON PLAN-FINDING

E.1 A PRECISE STATEMENT OF THE TOP-LEVEL LOOP

The top-level loop described below is for straight-line code. An extension for round-structured programs is discussed in section E.3.

1. Create the initial partial plan on the basis of any user, annotator or debugger advice.

2. For each active plan, explain the next statement of the program. Precisely what to do for each type of statement of code that may be encountered is stated in the next section.

3. After explaining a statement of code, update the unassigned model parts, list of violated and satisfied model statements, and list of violated and satisfied expectations.

4. Recompute the plausibility number of the new partial plans and rechoose the active subset, i.e. those plans with the highest plausibility number. The Plausibility Number is

```
            (+ (plausibility # of interpretation)
               (plausibility # of expectations))

  where (plausibility # of interpretation)
      = #satisfied.model.statements - #violated.model.statements

  and (plausibility # of expectations)
      = #satisfied.expectations - #unsatisfied.expectations
```

5. If a partial plan has either explained all the model parts or assigned a purpose to every statement of the program, then stop; else go to step 2.

Internal violations are more significant than external ones and should therefore probably be given more weight: however, the plausibility estimate has not been fine-tuned to that extent.

E.2 EXPLAINING AN INDIVIDUAL CODE STATEMENT

The following outline details what action the plan-finder takes as it examines a statement of the program.

CODE IS A SUB-PROCEDURE

A. Context: There are remaining unassigned model parts and a new main step is expected.

1. (PURPOSE (ACCOMPLISH <P>) where P is an unassigned model part. Prefer those model parts suggested by planning expectations. This constraint is usually insufficient to prevent some branching; therefore each of the resulting alternatives is pursued. The plan-finding process recurses to explain the code of the sub-procedure in terms of the model description for P. Those partial plans which result in a non-minimal number of violations when compared with their brethren are hung, i.e. they are not pursued further but are stored. They are reactivated only if the active partial plans themselves become implausible.

B. Context: In an open-coded sequence for model part P.

1. (PURPOSE (ACCOMPLISH <Q>)) where Q is the sub-part of P next expected.

2. (PURPOSE (ACCOMPLISH <Q>) where Q is not a part of P. This represents a Surprise. Demons are generated to await the completion of P in subsequent code.

C. Context: All model parts are assigned.

1. (PURPOSE CLEANUP). Cleanup to previous or canonical state.

2. (UNKNOWN PURPOSE). Pass on as an "extra-part" bug to debugger.

3. Plan incorrect. Undo previous planning choice.

CODE IS A TURTLE PRIMITIVE (FD, BK) ACCOMPLISHING A VISIBLE VECTOR

A. Context: Expecting new main step. The previous main-step has just been completed.

1. (PURPOSE (BEGIN OPEN-CODED SEQUENCE FOR <P>)), where P is an unassigned model part. Branching occurs in the choice of P.

2. (PURPOSE (SETUP POSITION)). This interpretation implies either a pen bug or a "retrace" bug since preparatory steps are required to be invisible. It is up to the debugger to decide the underlying

cause. The plan-finder simply reports the purpose of the statement as a "setup".

3. (PURPOSE (PIECE I <P>)), where P is a previously assigned model part. The code is accomplishing a part of P. This is decided by recognition of the code by a previously created planning demon.

B. Context: In the open-coded sequence for P, sub-part Q is expected.

1. (PURPOSE (BEGIN OPEN-CODING FOR <Q>)). This explanation is accepted if such an interpretation does not produce violations. However, if this binding does imply violations, then either Suspicion Demons are created or one of the alternative purposes below is accepted.

a. (Suspicion Analysis) Interpreting the current code as the beginning of an open-coded sequence for Q has generated violations. The suspicion is that one of the previous purpose assignments is incorrect. Some model part previously thought to be accomplished completely has, in fact, only partially been achieved. The particular violation suggests which model part is incomplete, e.g. the shorter of a set of unequal vectors. Generate a Suspicion Demon looking for code which will complete the part and eliminate the violation.

2. (PURPOSE (BEGIN OPEN-CODING FOR <R>)), where R is an unexpected model part. R is a surprise: the code begins a new model part. A demon is created to recognize completion of the expected part Q in subsequent code.

3. (PURPOSE (SETUP POSITION)). This interpretation implies a visibility bug.

4. Plan incorrect. Reject previous choice.

C. Context: All model parts have been explained.

1. (PURPOSE (ACCOMPLISH (PIECE I <P>)), where P is a previously assigned model part. This explanation is generated only if a previously created demon accepts the current code as satisfying its suspicion.

2. (PURPOSE UNKNOWN). Accept plan as correct. Code therefore has a bug -- possibly either a visibility bug or extra code. It is up to the debugger to decide.

3. Plan incorrect. Reject previous planning choice.

CODE IS A TURTLE PRIMITIVE (FD, BK) ACCOMPLISHING AN INVISIBLE VECTOR

1. (PURPOSE (SETUP POSITION)). If the invisibility is due to the vector being a "retrace", this is noted by "(SETUP POSITION BY

RETRACE)".

2. (PURPOSE (CLEANUP POSITION)). If the code returns the turtle to a previous endpoint, then that endpoint may represent a local Home state.

3. (PURPOSE (ACCOMPLISH <model part>)). This interpretation implies the bug that the code should accomplish a visible vector.

CODE IS A ROTATION (RT, LT)

1. Ordinarily, rotations are analyzed as SETUPS or CLEANUPS.

2. The exception is that if they are named explicitly as model parts (as in the TRIANGLE model), then they are formally treated similarly to vectors, i.e. they can accomplish parts or begin open-coded sequences for parts.

E.3 FINDING ROUND PLANS

The top level algorithm for directing the plan-finding analysis would require the following additions to handle round-structured programs. (Round plans are described in section 2.9).

A. Explain the control structure. Discover the end test, increment function, initial value for counter, and scope of round. Use structural analysis to identify as many parts of the control structure from the definition as possible.

B. Sub-routinize the basic round. Explain the round via the generic model parts. Use the algorithm described above to find the tie between the round and the model description of the generic parts.

Bibliography

[Floyd 1967]
Floyd, R. W.
"Assigning Meaning to Programs"
Proc. Symp App. Math AMS vol. XIX (1967)

[Fahlman 1973]
Fahlman, Scott
A Planning System For Robot Construction Tasks
AI-TR-283, MIT-AI-Laboratory (May 1973)

[Goldstein 1972]
Goldstein, Ira P.
LISP-LOGO - An Implementation of LOGO in LISP
LOGO Memo 7, MIT-AI-Laboratory (November 1972)

[Goldstein 1972]
Goldstein, Ira P.
GERMLAND
LOGO Working Paper 7, MIT-AI-Laboratory (February 1973)

[Hewitt 1971]
Hewitt, C.
"Procedural Embedding of Knowledge in PLANNER"
Proc. IJCAI 2 (Sept 1971)

[Hewitt 1972]
Hewitt, C.
Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot
AI-TR-258, MIT-AI Laboratory (April 1972)

[Hewitt 1973]
Hewitt, C., P. Bishop, and R. Steiger
"A Universal Modular Actor Formalism for Artificial Inteligence"
Proc. IJCAI 3 (Aug 1973)

[McDermott 1972]
McCermott, D.V. and G.J. Sussman
The CONNIVER Reference Manual
AI Memo 259 MIT-AI Laboratory (May 1972) (Revised July 1973)

[McDermott 1973]
McDermott, D.V.
Assimilation of New Information by a Natural Language Understanding System
Master's Thesis MIT-AI Laboratory (Feb. 1973)

[Moon 1973]
Moon, David, Reed, David et. al.
MACLISP REFERENCE MANUAL
Project MAC Memo, (December 1973)

[Naur 1967]
Naur, P.
"Proof of Algorithms by General Snapshots"
BIT 6, 1967, 310-316.

[Papert 1971a]
Papert, Seymour A.
"Twenty Things to Do with a Computer"
AI Memo 248, MIT-AI Laboratory (June 1971)

[Papert 1971b]
Papert, Seymour A.
"Teaching Children to be Mathematicians vs. Teaching About Mathematics"
AI Memo 249, MIT-AI Laboratory (July 1971)

[Papert 1971c]
Papert, Seymour A.
"A Computer Laboratory for Elementary Schools"
AI Memo 246, MIT-AI Laboratory (October 1971)

[Papert 1972a]
Papert, Seymour A.
"Teaching Children Thinking"
Programmed Learning and Educational Technology, Vol.9, No.5 (Sept1972)

[Papert 1972b]
Papert, Seymour A.
"On Making a Theorem for a Child"
Proc. ACM Conference (August 1972)

[Ruth 1973]
Ruth, G.R.
Analysis of Algorithm Implementations
MIT PhD Thesis (October 1973)

[Sussman 1970]
Sussman, G.J., T. Winograd, and E. Charniak
Micro-Planner Reference Manual
AI Memo 203, MIT-AI Laboratory (July 1970)
(revised December 1971)

[Sussman 1972]
Sussman, G.J. and D.V. McDermott
"From PLANNER to CONNIVER - A Genetic Approach"
FJCC (1972)

[Sussman 1973]
Sussman, G.J.
A Computational Model of Skill Acquisition
AI-TR-297, MIT-AI laboratory (August 1973)

[Sussman 1973]
Sussman, G.J.
"A Scenario of Planning and Debugging in Electronic Circuit Design"
AI Working Paper 54 (December 1973)

[Winston 1970]
Winston, P.H.
Learning Structural Descriptions from Examples
AI-TR-231, MIT-AI Laboratory (September 1970)

CS-TR Scanning Project
Document Control Form

Date : 4 / 30 / 96

Report # AI-TR-294

Each of the following should be identified by a checkmark:
Originating Department:

[✓ Artificial Intellegence Laboratory (AI)]
☐ Laboratory for Computer Science (LCS)

Document Type:

[✓ Technical Report (TR)] ☐ Technical Memo (TM)
☐ Other: ____________

Document Information

Number of pages: 228 (235-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
[✓ Double-sided]

Intended to be printed as :
☐ Single-sided or
[✓ Double-sided]

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown [✓ Other: Copy]

Check each if included with document:

[✓ DOD Form] [✓ Funding Agent Form] [✓ Cover Page]
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: ____________

Page Data:

Blank Pages(by page number): ____________

Photographs/Tonal Material (by page number): ____________

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-228) UN#'ED TITLE PAGE, UN#'ED ABSTRACT, UN# ACKNOWLEDGEMENTS, UN# DEDICATION, 5-228
(229-235) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGTS(3)

Scanning Agent Signoff:

Date Received: 4 / 30 / 96 Date Scanned: 5 / 2 / 96 Date Returned: 5 / 8 / 96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-294	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Understanding Simple Picture Programs		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ira Goldstein		8. CONTRACT OR GRANT NUMBER(s) N00014-70-A-0003 (ARPA) C40708X (NSF)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, Virginia 22209		12. REPORT DATE September 1974
		13. NUMBER OF PAGES 228
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automatic Debugging
Automatic Programming
Computer-Aided Instruction
Computer Education
Problem Solving
Program Debugging

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A computer monitor MYCROFT is described that understands simple programs by generating commentary, discovering plans and debugging mistakes. The interplay between procedural and declarative knowledge is analyzed and a description of various planning paradigms and debugging techniques is provided.