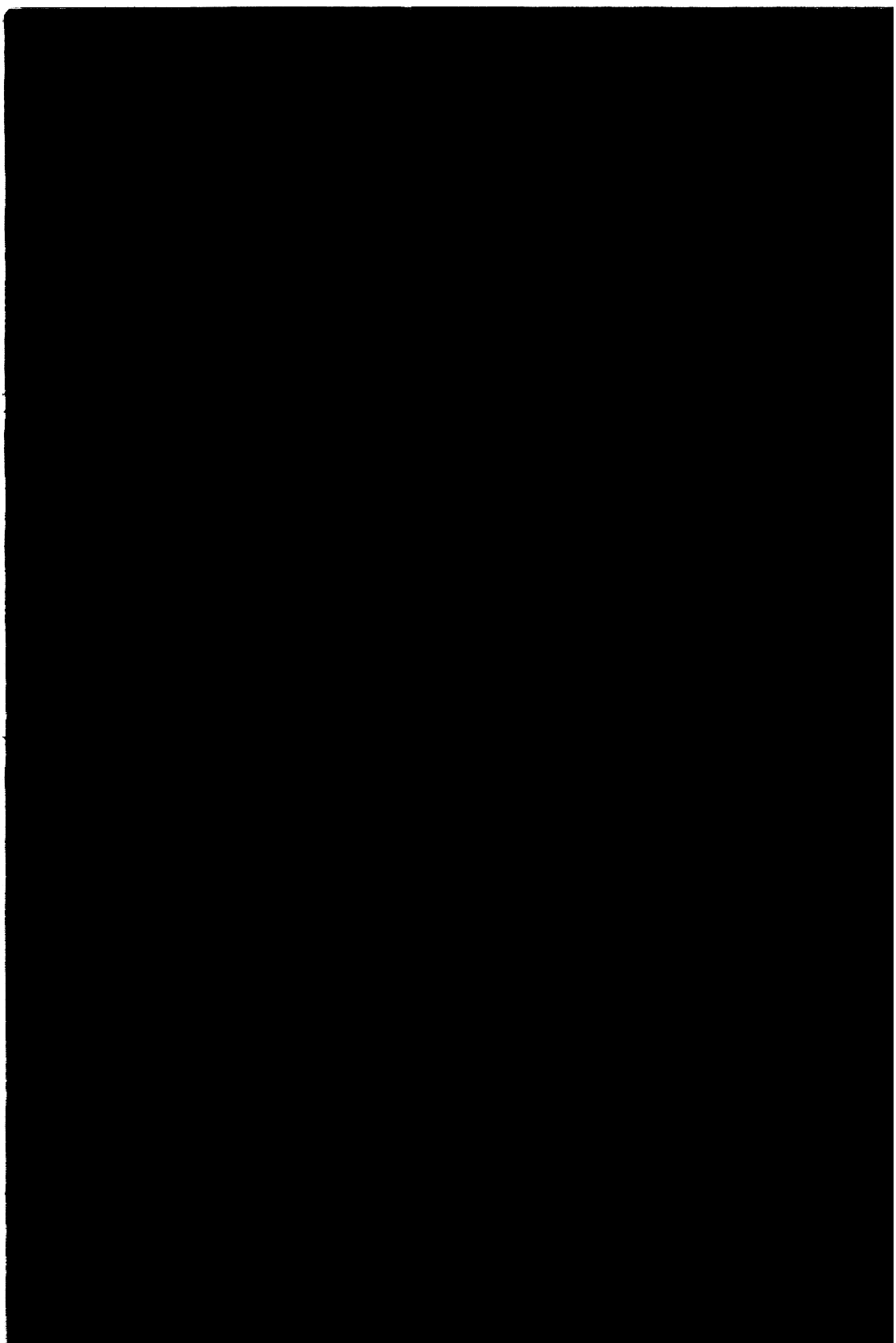


This blank page was inserted to preserve pagination.

PILOT: A STEP TOWARD MAN-COMPUTER SYMBIOSIS

by

Warren Teitelman

B.S., California Institute of Technology
(1962)

S.M., Massachusetts Institute of Technology
(1963)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF
PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF
TECHNOLOGY

September, 1966

Signature of Author.....Warren Teitelman.....
Department of Mathematics, June 14, 1966

Certified by.....Marvin Minsky.....
Thesis Supervisor

.....Herman Leeuwenn.....
Chairman, Departmental Committee
on Graduate Students

This empty page was substituted for a blank page in the original document.

PILOT: A Step Toward Man-Computer Symbiosis

by

Warren Teitelman

Submitted to the Department of Mathematics on **June 14, 1966**, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

PILOT is a programming system constructed in LISP. It is designed to facilitate the development of programs by easing the familiar sequence: write some code, run the program, make some changes, write some more code, run the program again, etc. As a program becomes more complex, making these changes becomes harder and harder because the implications of changes are harder to anticipate.

In the PILOT system, the computer plays an active role in this evolutionary process by providing the means whereby changes can be effected immediately, and in ways that seem natural to the user. The user of PILOT feels that he is giving advice, or making suggestions, to the computer about the operation of his programs, and that the system then performs the work necessary. The PILOT system is thus an interface between the user and his program, monitoring both the requests of the user and the operation of his program.

The user may easily modify the PILOT system itself by giving it advice about its own operation. This allows him to develop his own language and to shift gradually onto PILOT the burden of performing routine but increasingly complicated tasks. In this way, he can concentrate on the conceptual difficulties in the original problem, rather than on the niggling tasks of editing, rewriting, or adding to his programs. Two detailed examples are presented.

PILOT is a first step toward computer systems that will help man to formulate problems in the same way they now help him to solve them. Experience with it supports the claim that such "symbiotic systems" allow the programmer to attack and solve more difficult problems.

Thesis Supervisor: Marvin L. Minsky

Title: Professor of Electrical Engineering

ACKNOWLEDGEMENTS

The work herein was supported in part by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract number Nonr-4102(01), in part by the National Science Foundation Fellowship Program, and in part by Bolt Beranek and Newman, Inc., Cambridge, Massachusetts, under the Advanced Research Projects Agency, Contract No. AF19(628)-5065. Reproduction in whole or in part is permitted for any purpose of the United States Government.

I wish to express my gratitude to Marvin Minsky, for his supervision of this thesis, and to Seymour Papert and Oliver Selfridge, the other members of my thesis committee, for their critical reading of the manuscript. Oliver Selfridge's interest in the project and personal encouragement went far beyond the call of duty. It would be impossible to acknowledge all those not officially concerned with my thesis who both influenced and raassured me throughout the last three years. However, I want especially to mention Danny (Daniel G. Bobrow), who was always there when I needed him, and Claudia, who was never really away.

TABLE OF CONTENTS

Chapter		Page
Abstract		i
Acknowledgements		ii
List of Figures		iv
1	Introduction	1
2	Symbiotic Systems	9
3	The PILOT System	21
4	Facilities in the PILOT System	45
5	Experiments with a Question-Answering System	65
6	Experiments with a Problem Solver	93
7	Improving PILOT	139

Appendices		Page
1	Symbolic Differentiation in LISP	151
2	Using PILOT	153
3	List of Modifications	179
Bibliography		187
Biographical Note		193

LIST OF FIGURES

Number		Page
1	The function ADVISE	36
2	HISTORY	39
3	User-PILOT Interfaces	40
4	A Simple Problem Solver	95
5	Flow Chart of PILOT	156

CHAPTER 1

INTRODUCTION

The goal of artificial intelligence is to construct computer programs which exhibit the kinds of behavior that we call 'intelligent' when we observe it in human beings. These programs are usually so complex that the programmer cannot accurately predict their behavior. He must run them to see whether any changes should be made. Developing these programs thus involves a lengthy trial and error process in which most of the programmer's effort is spent in making modifications. PILOT is a system designed specifically to facilitate making modifications in programs. Examples of actual user-PILOT dialogue are presented.

This thesis is concerned with the problem of using computers more effectively for solving very hard problems, particularly problems in artificial intelligence.* These problems are extremely difficult to think through in advance, that is, away from the computer. In some cases, the programmer cannot foresee the implications of certain decisions he must make in the design of the program. In others, he can compare several alternatives only by trying them out on the machine. Since he cannot accurately predict the behavior of his program because of its size and complexity, he must instead adopt the more pragmatic policy of: "Let's run it and see what happens." The result is that solving these problems involves a lengthy trial and error process of "write some code, run the program, make some changes, write some more code, run the program again, etc.," even assuming

---

* For the definitive paper on Artificial Intelligence, see Minsky's "Steps Toward Artificial Intelligence" in Feigenbaum and Feldman (see bibliography.) This latter book also contains Minsky's bibliography on artificial intelligence as well as some of the more significant and interesting papers of recent years.

that the programmer does not make any "programming errors," which is rarely the case.

Moreover, in artificial intelligence problems, this process must often be prolonged beyond the debugging phase. It is important for the programmer to experiment with the working program, making alterations and seeing the effects of the changes. Only in this way can he evaluate it or extend it to cover more general cases.

Unfortunately, it is often not a simple matter to make changes in programs, especially large and complex ones. As a result, they frequently become "frozen," sometimes even before they are fully operational. Advances in programming languages have simplified the task of writing code. Time-sharing systems make the computer more accessible in the "run the program" phase. However, neither of these directly attacks this problem of making changes.

PILOT is a programming system that is designed specifically for this purpose. It improves and raises the level of interaction between programmer and computer when he is modifying a program. It takes over many of the chores of programming-debugging-changing, leaving the programmer to concentrate on conceptual problems. PILOT is not a static system; it can grow and evolve along with the programs the user is developing. As a result, it can be tailored to any particular user and any particular problem. In sum, PILOT helps the programmer to be more effective. This in turn enables him to attack and solve more difficult problems.

PILOT is written in LISP[1,32,33] and operates in the Project MAC time-sharing system at the Massachusetts Institute of Technology.[14,10] The next pages give some examples of actual user-PILOT dialogue. The user's inputs are in lower case and PILOT's responses in upper case.

In the first example, the user is experimenting with a version of the Deductive Question Answering System of Fischer Black.[2] In this system, there are direct statements, such as "at(I,home)" and "want(at(I,airport))," and conditionals, such as "smaller(x,y),smaller(y,z)→smaller(x,z)." Deductions are performed by substitution and by "detachment," which is a generalized modus ponens. Thus given the two direct statements "smaller(car,house)" and "smaller(dog,car)," and the above conditional, the program can deduce "smaller(dog,house)" by substituting dog for x, car for y, and house for z.

One difficulty with this scheme is that the deduction of even a "true" statement may not terminate because the same conditional may be considered repeatedly. This cannot be avoided in general, because of the existence of "undecidable" propositions; but in many cases it can be circumvented. In this example the user is experimenting with different methods to achieve this.

The most straightforward approach would be for the program to keep track of what it was doing. It could then apply some simple criterion to decide whether or not it was "looping," and if it was, to abandon the deduction. The danger with this is that a particular criterion might stop a deduction which would otherwise terminate - or it might not recognize one that would

not. But, this is the point of experimenting with a working program: to try out various ideas.

In our example, SOLUTION1 is the function that handles new "questions." The user wants to prevent it from entering itself more than twice recursively. In other words, this is the criterion he is considering.

```
(tell solution 1, (before number advice),
If (countf history ((solution1 -))) is greater than 2, then quit)
```

The user tells PILOT to modify the function SOLUTION1. The phrase "(before number advice)" tells PILOT to insert this advice immediately before the advice containing the key word "number." (This refers to advice the user has previously given SOLUTION1 - see Chapter 5 for complete experiment.) The user wants SOLUTION1 to cease computation whenever "(countf history ((solution1 -))) is greater than 2." COUNTF is a general counting function available in PILOT. Its arguments here specify a search through the current "history" of the computation counting the number of times the function SOLUTION1 has previously been entered. The "history" is provided and kept up to date automatically by PILOT.

```
TRANSLATION:      (EDIT SOLUTION1 BEFORE ((INSERT (ADVICE COND
      ((GREATERP (COUNTF HISTORY (QUOTE ((SOLUTION1 -))) HISTORY
) 2) (LIST
          NIL))
      (T NIL)) BEFORE NUMBER (BACKTO ADVICE) UP1) STOP))
```

This is the translation of the user's request. It has been interpreted as a call to the function EDIT. EDIT will INSERT at the correct place the piece of advice that is the translation

of "if (countf history ((solution1 --))) is greater than 2, then quit."

```
TRANSLATION:      (FLIP
      ($ NUMBER (BACKTO ADVICE) UP1 $)
      (1 2 3 (ADVICE COND
         ((GREATERP (COUNTF HISTORY (QUOTE ((SOLUTION1 -)))
 HISTORY) 2) (LIST
            NIL))
         (T NIL)) -2 -1))
(SOLUTION1 BEFORE)
```

This second translation takes place inside of EDIT. (INSERT ... BEFORE ...) has been transformed into a call to FLIP. FLIP will insert the advice at the correct point. Actually FLIP will construct a new list with this advice inserted. EDIT then replaces the old advice structure with the new one, and returns control to PILOT for more requests.

* * *

Later, the user considers a more sophisticated criterion for terminating deductions. For this, he needs a function to determine whether two questions are identical, except for substitution of variables. He defines a new function, MATCHES.

```
(define matches (x y) as
If x is null, then y is null,
If (car x) is equal to (car y) or
   (variable car x) and (variable car y),
      then (matches cdr x cdr y))
```

using PILOT to translate from his own language into LISP.

```
TRANSLATION:      (DEFLIST ((MATCHES (LAMBDA (X Y) (COND
         ((NULL X) (NULL Y))
         ((OR
            (EQUAL (CAR X) (CAR Y))
            (AND
               (VARIABLE (CAR X))
               (VARIABLE (CAR Y)))) (MATCHES (CDR X) (CDR Y
)))
         (T NIL))))) EXPR)
(MATCHES)
```

This is the translation of his request. Here instead of calling EDIT, PILOT calls DEFLIST, which defines MATCHES as a function (an EXPR) of two variables, X and Y. Note that PILOT, because it knows how many arguments a function takes, has correctly parsed (MATCHES CDR X CDR Y) as (MATCHES (CDR X) (CDR Y)), (VARIABLE CAR X) as (VARIABLE (CAR X)), etc.

* * *

One of the claims of PILOT is that it frees the user from having to consider the inner workings of his system. This is illustrated in the above example. Here the user has taken an unfamiliar system, written by a different person, and performed certain nontrivial modifications. This was done with only a superficial knowledge of the design and construction of this system.

However, it is when the user programs within PILOT, taking into account its capabilities, that the greatest returns are obtained. He can proceed almost directly from flowchart to working system, filling in the details using PILOT. Thus it is no longer necessary to complete the details of planning before commencing to program. The program can be developed on-line.

This is the case with our second example. The user has programmed a simple flowchart. Some of the functions even have null definitions, that is (LAMBDA NIL NIL). The following dialogue shows how he can modify his system to solve a problem new to it: the cannibal and missionaries problem. (Note that in the last line, the computer, with the line *T*, announces that it has, in fact, successfully solved the problem.)

```
solve (cannibal and missionaries)
 (DONT KNOW HOW)


(start with side1 (m m m c c c)  side2 nil, to side2, from side1)
 START

(tell goalp, return with side1 is null)
 GOALP

(tell moves, return with '((move1) (move2)))
 MOVES

(define move1 as alltran valueof from '($1) '((2) 1 3))
 MOVE1)

(define move2 as alltran valueof from '($1 $ $1) '((2 4) 1 3 5))
 (MOVE2)

(tell make, to (y) (setq y from) and bind (valueof ' from)
to (cdr move) and bind (valueof ' to) to (append car move valueof to)
and bind from to to and bind to to y)
 MAKE

(tell progress, if ' m is a member of side1 and ' m is a
member of side2 and (countq side1 ' m) is not equal to
(countq side1 ' c), then quit)
 PROGRESS

(gps : save (cons from side2) on hist)
 GPS

(after gps : pop hist)
 GPS

(tell progress, if searchf hist (((= from) $ / (setequal (= side2))))
then quit)
 PROGRESS


solve (cannibal and missionaries)
 *T*
```

These examples give the flavor of the interactions between the user and PILOT. It is not expected that the details of the dialogue will be self-evident. Remember that while there are many conventions used in communicating with PILOT, they are the user's conventions, in this case mine, and as such have intuitive meaning to me. Learning to use PILOT involves building a language for communicating certain operations. The above examples indicate, to some extent, the type of language I have found useful. If you were using the system, you could, and undoubtedly would, change the format of some or all of the operations specified to PILOT in these examples. This in part, is what makes PILOT symbiotic.

CHAPTER 2

SYMBIOTIC SYSTEMS

Man-computer symbiosis involves very close coupling of man and machines. This chapter describes several of the more successful "symbiotic" programming systems. While none of these perform operations for the man that he could not do himself, they allow him to operate at a greater level of abstraction, and thereby to concentrate more fully on the problem he is trying to solve. This in turn has a substantial effect on his productivity.

Symbiosis is a mode of living characterized by intimate or constant association or close union of two dissimilar organisms. The usual implication is that the association is advantageous to one or both.[18] There are many examples of symbiosis in both the botanical and zoological worlds, among these the symbiosis of algae and fungi (called lichens), ants and aphids, and the pilot fish and the shark.* But until 1960, the term symbiosis had only been applied in the biological context.

In 1960, Dr. J.C.R. Licklider introduced the term man-computer symbiosis in an often-cited paper by that name.[22] Concerning the problems involved in developing symbiotic systems he stated:

"Among the problems toward which man-computer symbiosis is aimed -- problems that men and computers should attack in partnership -- are some of great intellectual depth and intrinsic difficulty. The main problems that must be solved to bring man-computer symbiosis into being, however, appear not to be of that kind. They are not easy, but their difficulty seems due more to limitations of technology than to limitations of intelligence." [23] (italics mine)

---

* the latter symbiosis, that of pilot fish and shark, is part of the derivation for the name PILOT. The name is also meant to reflect the fact that this is a pilot system for man-computer symbiosis.

Much effort has been devoted to developing symbiotic systems in the few years since this statement was made. In these systems, the computer performs the routine work -- a surprisingly large percentage of the total amount -- that must be done to prepare for insights and decisions in technical and scientific thinking. Man sets the goals, performs the evaluations, and in general, guides the course of the investigation.

In evaluating these systems, one must realize that there are degrees of symbiosis. You can always improve a system. However, Licklider has set as a subgoal the development of "a mechanism that will couple man to computer as closely as man is now coupled to man in good multidisciplinary scientific or engineering teams." The systems described in the following pages certainly achieve this goal.

Sketchpad[46]

Most computers use keyboards for on-line input and output. This excludes the use of diagrams for communication with the machine. About 1960, an interest began to build up in developing computer display systems whereby man and computer could converse rapidly through the medium of line drawings. The most significant system to arise from this impetus was "Sketchpad," the work of Ivan Sutherland.[46] Using Sketchpad, the user could make two-dimensional sketches with a light pen directly on a computer CRT display, and then modify and move parts of the drawing around as he wished. Sketchpad would preserve the topology of the drawing and carry out computation on the figures so drawn. For example, in one mode, when the user drew a line, the computer would draw an absolutely straight line. When the user made two lines come almost together at a corner, the computer would make

them come exactly together, etc. Furthermore, Sketchpad would remember that the lines were joined so that if the operator moved one of the lines, it would move the other one in such a way as to maintain the intersection at the corner.

In other modes of operation, Sketchpad would make perfect arcs, straighten up figures so that nearly horizontal lines were made exactly horizontal and nearly vertical lines were made exactly vertical. It would remember the shape of a figure or subfigure so that the user could request replicas of this figure at various points of the diagram. Sketchpad thus permitted the user to make an assembly of several elementary figures, to replicate assembles, to make assemblies of assemblies, etc.

In one impressive demonstration, Dr. Sutherland sketched the girder of a bridge, and indicated the points at which members were connected together by rivets. He then drew a support at each end of the girder and a load at its center. The sketch of the girder then sagged under the load, and a number appeared on each member indicating the amount of tension or compression to which the member was being subjected.

Sketchpad has been extended to three dimensions by Johnson.[20] In Sketchpad III, the user can add a line to a plan and have it appear simultaneously in the front field, the side view, and the oblique representation. When he rotates the oblique representation, the orthogonal views change appropriately, etc.

Sketchpad is primarily a research system; no one today is <u>using</u> Sketchpad. However, the insights gained during its development,

and the psychological impact of the program itself have greatly influenced the construction of symbiotic systems, especially those involving graphical input and output.

Computer-Aided Circuit Design

Another graphical research program involves the on-line construction of electrical networks.[41] An electronic circuit designer interacts directly with the computer through a typewriter and CRT graphical input-output equipment. He builds his circuit by keying in an element at a time to the computer, placing the light pen on the CRT to show where it goes. In this way, he can compose on the screen any circuit he wishes; then he can ask the computer to analyze it.

The most significant consequence of this man-machine interaction, as with the other systems described, is the short time, usually on the order of seconds, between a user request and the computer response. Lindgren[26] states: "the 99.99 percent of engineers who are designing circuits without on-line graphical-language facilities, are, in one sense, already 'living in the past.'"

* * *

One of the most obvious areas in constructing symbiotic systems is mathematics, since mathematical tasks are usually better defined than those in other fields. Many "mathematical laboratories" have been developed to provide the mathematical scientist with the services of an on-line computer. Some of these are described below.

The Symbolic Mathematical Laboratory[29]

One of the problems a user performing any realistic mathematical computations soon encounters is the inadequacy of the keyboard for communication with the machine. Consider the following expression:

$$R_0(\omega)=\frac{2\cdot\ell}{\pi}\cdot\frac{\log\omega}{a+1}+\frac{2\cdot a\cdot\omega}{\pi\cdot(\log\omega)}\int_0^{\pi}\theta_1(t)\cdot M_{a-1}(\omega\cdot t)\,dt$$

For writing such expressions, the mathematician employs a large character set, and utilizes subscripts and superscripts extremely liberally. He observes certain conventions concerning the physical size, grouping, and placement of subexpressions. All of these make it easier for him to read and comprehend mathematical formulae. Even if a keyboard could be designed to handle expressions of the above type, it would have to be unreasonably large and complex. In addition, how are subexpressions to be referred to? The mathematician can point to them or in other ways refer to them directly, when he is working on paper or blackboard. Requiring him to input a subexpression each time that he wishes to refer to it would make for a very unsatisfactory system.

The Symbolic Mathematical Laboratory[28,29,37] is a system designed to solve these problems. In the original proposal[37] Minsky describes a program "for displaying publication-quality mathematical expressions given symbolic (list-structure) representations of the expressions." The goal is to produce "portraits" of expressions that are sufficiently close to conventional typographical conventions that mathematicians will be able to work with them without much effort -- so that they do

not have to learn much in the way of a new language, so far as the representation of mathematical formulae is concerned."[37]

"We imagine that the user is engaged in performing a mathematical exploration. For example, he might be trying to find a solution to a differential equation. At the moment, he has displayed on the screen one or two equations, and he has in his head the name of several other expressions or partial results already studied and filed away. He decides to perform one action, e.g., substituting a displayed equation, solving it for some variable, expanding some subexpression in a certain way, or perhaps simply displaying something else. This action is requested by some combination of light-pen and keyboard signals. These signals are encoded and transmitted to LISP, which computes or retrieves the required new expressions and transmits them back to the display system. The latter then compiles and displays the desired new picture."[37]

The basic ingredient of this system is the program sequence that converts an internal mathematical expression into a conventional printed representation. Martin uses a Polish prefix notation convenient for LISP operations to represent expressions internally. For example, (PRD (PRD 2 *L (PWR PI -1 NIL) NIL) (PRD (LOG OMEGA NIL) (PWR (PLS *A 1 NIL) -1 NIL) NIL) NIL) is the internal representation of $\frac{2.\ell}{\pi} \cdot \frac{\log \omega}{a+1}$.* Since the correspondence between internal representations and wnat is being displayed is maintained by the program, the user can refer to any particular subexpression, by pointing at it, and the program selects and operates upon the corresponding internal structure.

The converse problem of converting the external printed representation to internal representation has not been treated as extensively in Martin's program, although he intends to add a character recognition scheme based on ARGUS[47] for direct input from the CRT. However, it is not as serious as the display

* The $R_0(\omega)$ expression on the previous page is an actual example. See [28] for a photograph of this expression as it appears in his system.

problem, because it is not done as often -- since most of the expressions used by the mathematician will either be generated by the program or be subexpressions of expressions already in the system. Therefore, the user can tolerate entering expressions by some tedious, more conventional keyboard method, especially since he can see the displayed expression as it goes in and correct the computer if it, or he, has made any mistakes.

Other Mathematical Systems

D. Maurer has designed a system for a more sophisticated mathematician, specifically the algebraist.[30] His program is conversant in such subjects as groups, subgroups, ideals, etc., and can respond to requests of the form: generate the set of all normal subgroups of a particular group; generate subsemigroup z from element x of y; etc. Maurer has preprogrammed many of the operations needed by the algebraist, and has included facilities for introducing new ones as needed. However, the system has not yet been put to practical use.

MATHLAB[13] is a LISP program which emphasizes continually increasing powers. MATHLAB can formally integrate certain functions, differentiate, factor, expand, simplify, etc. Since it is written in LISP, new operations can be added very easily. MATHLAB is currently operating on the Project MAC time-sharing system.

CALCULAID and MAP are two more systems for using the computer as a mathematician's helper. CALCULAID[50] is oriented towards writing programs to solve large problems with much data. It has built in FIT and REGRESSION operators, and a convenient way of specifying matrix operations. MAP[21] has facilities for

performing convolutions, Fourier transforms, and other more sophisticated analytical operations. In MAP the user is encouraged to consider himself as conversing with the computer, which then performs the operations. This is in contrast to CALCULAID, where the system is not viewed as an agent so much as a collection of useful subroutines, easily available.

MUSIC Laboratory*

Perhaps at the other end of a spectrum is an attempt to create an environment, on the computer, which is conducive to the composition and analysis of music. Using the computer as an expensive instrument is not a new idea. In 1961, Peter Samson wrote a music compiler for the Digital Equipment Corporation PDP-1 computer.[43] The basic idea was that the user would encode the musical score into a series of numbers, each note being denoted by two numbers - one for its pitch, the other for its duration. The computer would then play the music, utilizing its digital-analog converter to control the voltage on a speaker directly. Thus the computer would play a middle C by varying the voltage 256 times a second, essentially building its own square wave. The computer was even fast enough to construct in real time the wave form corresponding to a three part harmony. However, the MUSIC Laboratory project currently underway at M.I.T. has even more ambitious goals.

The standard teletype of the DEC PDP-6 has been augmented by an 88 key piano keyboard which is connected directly to the computer. Thus the user can play a melody, hear what it sounds like - as performed by the PDP-6 - and also see the score displayed visually on the scope. He can then edit the score, using

* No documentation is available.

the light pen, the teletype, or the piano keyboard, and hear it played again. Programs are being written to allow the user to request the computer to fill in a harmony to a particular melody, or to construct variations on a theme and to play them back to the user.

Synergetic Systems

The most important point about the systems described above, a point which also applies to PILOT, is not so much that they are symbiotic, i.e., cooperative, as that they are synergetic. Synergism is the cooperative action of discrete agencies such that the total effect is greater than the sum of the two effects taken independently. An example of this is the action of penicillin and streptomycin when taken together. The extreme potency of the combination of tranquilizers and alcohol presents another, more familiar example.

The most significant aspect of the systems described above is the synergetic action of man and machine that they foster. Close examination of these programs reveal that they do not, in themselves, do anything remarkable, nor do they represent any significant advance in sophistication. Computer programs that analyze circuits or invert matrices in the course of solving a problem are not uncommon. However, there is a substantial effect on the productivity of a man if he can immediately substitute an expression for a variable and integrate. The mere fact that he could have performed each individual operation himself is not important, nor does it affect the synergetic quality of the interaction. What is important is that the overhead involved in switching tasks is eliminated, or at least substantially reduced. Thus the user can operate at a greater level of

abstraction and thereby concentrate more fully on the problem itself.

This same phenomenon occurs with the so-called higher level programming languages. These languages do not do anything for the programmer that he could not do himself. In other words, you could program everything in machine language directly. However, the fact of the matter is that suitable programming languages do allow the programmer to attack and solve much more difficult problems. As an example, ten years ago an M.I.T. graduate student in electrical engineering received a master's degree with a thesis (program) for performing symbolic differentiation. This same feat can be duplicated today in a half dozen lines of LISP coding.* The point is not that LISP makes it easier to solve problems, but that thereby LISP makes it possible to solve harder problems. In this particular example, the amount of effort required to construct a differentiation routine in LISP was comparable to that required for a small subroutine. This is where the synergetic effect enters because now the programmer can build systems in which this differentiation routine is precisely that: just a small subroutine (as it is in the systems of Martin and Engelman).

The question here is one of human limitations. Once the programmer has constructed and debugged a differentiation routine, it should not matter whether it was written in six lines of LISP or five thousand machine instructions. In practice, however, there is a limit to the size and complexity of a system that one person can successfully construct, assuming that he is starting from scratch. Unfortunately, with artificial intelligence pro-

---

* See Appendix 1.

grams, this limit is frequently encountered while there are still ideas remaining to be tried.

The PILOT system represents an exercise in applied synergism that parallels and complements that of high level programming languages. We might draw the analogy that PILOT is to an editing program what high level programming languages are to machine code. PILOT does not do anything for the user in the way of making changes that he could not do himself by editing or rewriting. But the fact that PILOT does do it means that the user does not have to. As with the systems described earlier, he is free to operate at a much higher level of abstraction and unencumbered by bookkeeping. He thus finds himself able to solve problems he could not even consider before. This is what makes user-PILOT a synergetic system.

This empty page was substituted for a blank page in the original document.

CHAPTER 3

THE PILOT SYSTEM

The function of PILOT is to allow the programmer to treat his program as if it were a block diagram. This places certain requirements on PILOT in terms of the structure of programs, data in programs, and modifying programs. This chapter presents a model of programs and programming that emphasizes how a program looks to its author. The basic building blocks of programs in the model are procedures, and the operation of advising consists of modifying the interfaces between these procedures. Implementation of a system that permits advising is described within the LISP programming system. Viewing the entire system of user-PILOT-programs as one program, it is possible to modify the interface between the user and PILOT to permit more flexible interaction, as well as modifying the interface between PILOT and the user's program to allow more complex types of advice to be specified.

One of the most useful ways of describing and representing a computer program is the block diagram. In it, the individual processes that take place inside the program are clearly isolated. Furthermore, it permits either elaboration of the details of some part of the computation, or bypassing details (by merely drawing a small rectangle and labeling it PROCESS). It is valuable in planning a program, because it makes it easy to see the flow of control and the interactions between various parts of the program. Moreover, a program in this representation can easily be modified, e.g., move blocks from one point to another, change lines of communication, add new blocks, replace old blocks, etc.

Unfortunately, computer programs tend to lose the nice features of block diagrams once they are written as a sequence of instructions.

The function of PILOT is to allow the programmer to continue to treat his program as if it were a block diagram. This places certain requirements on PILOT in terms of the structure of programs, data in programs, and modifying programs. These are discussed below.

Structure of Programs

One of the principal advantages of block diagram representations is their flexibility. They do not require him to be consistent about the amount of detail from diagram to diagram. If it seems appropriate to the programmer to describe a certain section of his program in great detail, while only sketching briefly some other portion -- for whatever reason he may have -- he can easily do this. Furthermore, he can represent the same program in different ways at different times; he is not compelled to make one choice and be bound by it.

If this flexibility is to be captured in PILOT, the system cannot restrict the user to some narrow range of preconceived structures. With respect to describing and representing programs, PILOT should enable the user to maintain a wide range of choice. Regardless of objective criteria for choosing one representation over another, the user must be allowed to choose whatever structure seems the most convenient or desirable to him. In other words, he must be allowed to make a subjective choice.

Subroutines

The standard way of structuring a program (as opposed to a block diagram) is by means of the subroutine. Programmers use subroutines to make their programs look more like their block

diagram representation.* This makes constructing and debugging a program much easier. Subroutines in a program are the analogue of the blocks in the block diagram, and, to a certain extent, their use retains many of the advantages of the block diagram. For example, to move a subroutine from one place to another in the program, all that is necessary is to move the call to the subroutine - usually only one or two instructions. To insert a subroutine, all that is necessary is to insert a call (assuming, of course, that the subroutine has been written). In the same way that blocks can be treated as separate entities, it is often possible to treat subroutines as separate from the rest of the program, and to construct and modify them accordingly. Thus, at least to the level of the subroutine, programs can be treated as block diagrams.

However, below this level, rigor mortis sets in. The individual blocks correspond to the way the programmer partitions the task, and the subroutines correspond to these blocks. But he may change his mind. What was viewed as a single operation initially may at some later point best be considered as three or four distinct operations. Remedying this in the block diagram is simple: replace the block by several smaller blocks. However, breaking a subroutine into three or four smaller sections is often not that easy. And yet frequently the programmer must be able to deal with units smaller than the subroutine.

Procedures

The "atomic" unit of structure in my model of programming will be the procedure, not the subroutine. A procedure is

* Other considerations such as computation time, and program space also affect the use of subroutines.

defined as a collection of n entrances and m exits together with input-output characteristics. This definition purposely does not require a procedure to be any easily isolated part of the program. If, of course, a procedure is a subroutine, identifying it is simplified. However, a procedure may be a part of a subroutine, or even parts of several subroutines. Essentially, a procedure is a chunk of code that the programmer wants to treat as a single unit. PILOT enables him to do so.

Data in Programs

Procedures are defined in terms of what they do, that is in terms of transformations on certain variables. These variables are called essential variables. Essential variables are not the only variables that are altered by a procedure. For example, in a time-sharing environment, the state of certain disc and drum variables (registers) may change thousands of times while executing a program. Even if we consider only variables specifically utilized in or changed by the operation of a program, many of these will be low-level, or local variables, and thus not important to the programmer. Describing the state of the computer at any time during a computation in terms of essential variables is more in keeping with the block diagram.

Essential variables are similar to the arguments of a subroutine. However, in many subroutines the essential variables are not passed through the calling sequence. Furthermore, procedures need not be subroutines, nor have a specific call. Thus the data used by the procedure may be scattered throughout the program. However, it must be available to the procedure. Some information is not available to a procedure. For example, the only variables that may be referenced inside of a FORTRAN subroutine, besides the arguments to the subroutine, are those specifically declared to

be COMMON. Some information may not be available to the program at all. For example, information regarding the function that originated the call to a particular FORTRAN subroutine is not, in general, available anywhere within the program. (Of course, the programmer can specifically provide this information by including the name of the function as one of the inputs to the subroutine in question.)

Variables that are available but non-essential are called extraneous. In many programming languages, there can be no extraneous variables -- everything is either mentioned or else not available.* (At the level of machine language, of course, everything is available.) This immediately precludes the computation of the name of a variable, i.e., indirect reference to it.

Extraneous variables are important because they may at some time become essential to some procedure, as a result of program modification. If they are not available, they cannot be used. PILOT automatically makes available information regarding what is happening "above," i.e., what functions have been called, what their essential variables are, etc., so that the programmer does not have to foresee explicitly what information he will need in a particular procedure.

Modifying a Program

There are two ways a user can modify programs in this subjective model of programming: he can modify the interface between

* There are exceptions. In LISP 1.5, uncompiled functions have their arguments bound on the ALIST so that in any particular function, all of the essential variables of functions entered previously are available. Similarly, in COMIT,[50] the 127 shelves are available, but often are extraneous variables. But, by and large, the above statement is true.

procedures, or he can modify the procedure itself. (Since procedures are themselves made up of procedures, modifying a procedure at one level may correspond to modifying the interface between procedures at a lower level.) Modifying the interface between procedures is called advising. Modifying a procedure itself is editing.

Advising is the basic innovation in the model, and in the PILOT system. Advising consists of inserting new procedures at any or all of the entry or exit points to a particular procedure (or class of procedures). The procedures inserted are called "advice procedures" or simply "advice." Since each piece of advice is itself a procedure, it has its own entries and exits. In particular, this means that the execution of advice can cause the procedure that it modifies to be bypassed completely, e.g., by specifying as an exit from the advice one of the exits from the original procedure; or the advice may change essential variables and continue with the computation so that the original procedure is executed, but with modified variables. Finally, the advice may not alter the execution or affect the original procedure at all, e.g., it may merely perform some additional computation such as printing a message or recording history. Since advice can be conditional, the decision as to what is to be done can depend on the results of the computation up to that point.

The principal advantage of advising is that the user need not be concerned about the details of the actual changes in his program, nor the internal representation of advice. He can treat the procedure to be advised as a unit, a single block, and make changes to it without concern for the particulars of

this block. This may be contrasted with editing in which the programmer must be cognizant of the internal structure of the procedure.

In the PILOT system, both of these facilities are available. Considerable effort has been devoted to providing the user with a sophisticated editor, with expandable syntax and semantics, in order to match the flexibility of the advice-giving mechanism. The editor allows the user to specify structural changes conveniently, while the advisor handles interface modifications. The advisor is usually more convenient, since it handles more of the details. However, the user may wish to perform what could be an interface modification by editing the procedure itself, possibly because of efficiency. In fact, for certain types of operations, the advisor itself uses the editor.

It is clear that both advising and editing complement each other, and that both are needed to ensure the programmer freedom to treat his program in ways that seem desirable to him. The choice of which of the two facilities he wishes to use for a particular operation is a matter of his personal preference, and depends on the nature of the change.

Class Modifications

It is most important that the user be able to modify a class of procedures, as well as individual procedures, i.e., to refer to procedures associatively as well as nominally. Until now we have assumed that the procedure to be modified had already been identified and located, but this is not necessarily the case. For example, the user may wish to specify changes to a class of procedures in which certain members have not yet been defined.

Alternatively, the decision of whether or not a modification applies to a particular procedure may have to be postponed until the procedure itself is actually entered. In the former case, it will be necessary to monitor the definition of new procedures in order to make the appropriate modifications. In the latter case, it may even be necessary to require all procedures (or a sufficiently large class of procedures) to inquire at the time they are called whether or not there are any modifications that should affect them. In both cases, it is not possible to locate procedures that are to be modified at the time the user specifies the modification.

Implementation

It is clear that implementing PILOT will be greatly facilitated by an appropriate choice of programming language. We must avoid translators, assemblers, and compilers that assume that the programming will be completed before the translation is begun, and that the program will not actually be run until all the assembling and compiling has been finished. In languages of this type, FORTRAN, COMIT, MAD, etc., it is difficult to write programs that construct or modify procedures because the communication between procedures is so deeply embedded in the machine instruction coding, that it is very difficult to locate entrances, exits, essential variables, etc.

The language I have chosen to use is LISP 1.5.[32,33,1] The LISP formalism is convenient for programming recursive tasks, which makes it good for problem solving and other heuristic programs. It is a list processing language, which is a necessity for programs of this type because storage allocation requirements cannot be predicted prior to run time, as the size and structure

of the data are determined by the computation. LISP is well suited to symbol manipulation, which means that it is possible to talk about the names of variables, and perform computations which produce them. Finally, I chose LISP, over IPL or SLIP for example, which also possess several of the attributes above, because I am familiar with LISP and find it convenient to program in the functional notation it provides.

In LISP, all data are in the form of symbolic expressions, or S-expressions. S-expressions are of indefinite length and have a branching tree structure in which subexpressions can be readily isolated. LISP computations are also written in the form of S-expressions. This makes LISP especially adaptable for our purposes. Like machine languages, and unlike most other higher level languages, one can write programs in LISP which will generate programs for further execution. Furthermore, it is possible to execute data as programs, and conversely treat programs as data.

This suggests an easy way of implementing advising: define a LISP function, ADVISE, which treats as data the advice to a procedure and the procedure itself, and executes the procedure with the appropriate modifications. By giving a name to each procedure that is advised, we create a canonical place where information associated with the procedure can be stored: its property list. The definition of the procedure, and the advice associated with it can be stored on and retrieved from its property list by the function ADVISE. Thus ADVISE requires only the name of the procedure, and the name of the entry or exit of the procedure. The operation of advising a procedure is therefore reduced to locating its entry and exit points, and replacing

them with a call to ADVISE, specifying the name of the procedure, the name of the entry or exit. The advice is stored on the property list of the name of the procedure, and the corresponding modifications are automatically performed when ADVISE is called.

The actual definition of the function ADVISE is not this general. The current implementation imposes the restriction that only one entry and exit may be allowed. This is because the effect of multiple entries and exits can be achieved within the current implementation, and because it is questionable whether the greater generality would justify the extra effort.

Multiple Entries and Exits in LISP*

The notation of LISP is function oriented. It encourages the user to define different functions for different tasks, especially because LISP makes it easy to call functions, and to nest sequences of function calls. Each function call in LISP has a single, canonical entry and exit, namely that provided by the LISP interpreter or compiler. The user normally does not concern himself with entries and exits; instead he thinks in terms of inputs (arguments) and outputs (values). The only exception to this occurs within the special form "PROG."

The PROG feature in LISP allows one to write ALGOL-like programs containing a sequence of LISP statements to be executed.

* This discussion presumes some familiarity with the LISP notation.

This is a concession to the fact that certain tasks are easier when not expressed in functional notation.*

In a PROG, the programmer can explicitly control the flow of computation by using labels and GO statements. For example, the function LENGTH defined without using a PROG is:

```
(LAMBDA (X) (LENGTH1 X 0))
```

where LENGTH1 is defined as:

```
(LAMBDA (X Y) (COND
((NULL X) Y)
(T (LENGTH1 (CDR X) (ADD1 Y)))))
```

Here using a PROG results in a more natural definition:

```
(LAMBDA (X) (PROG (U V)
(SETQ V 0)
(SETQ U X)
A (COND ((NULL U) (RETURN V)))
(SETQ U (CDR U))
(SETQ V (ADD1 V))
(GO A) ))
```

It is only inside a PROG that the LISP programmer can effect multiple entries and exits, namely by entering or leaving a procedure, i.e., a collection of LISP statements, at different labels. Multiple entries and exits from LISP functions are simulated by transmitting extra information in the calling sequence or value of the function. For example, in machine language programming

---

* PROGS are also used because they produce more efficient computations when compiled then the corresponding recursive definitions. This occurs because it is not necessary to rebind all of the arguments of the function on the push-down list for each iteration of the process. For this reason, experienced LISP programmers occasionally use PROGS even when a recursive definition would be more natural and intuitive.

it is often common practice to write the trigonometric functions as one subroutine with different entrances. This could be done in LISP by defining TRIG as a function of two variables, X and Y, where X was either SIN, COS, TAN, etc., and have the appropriate routing performed inside TRIG. Since it is so easy to transmit extra information in LISP, this is usually the way it is done, especially since there is an advantage in having separate operations, or procedures, correspond to separate functions: many facilities such as TRACE, BREAK, COMPILE, are oriented around functions.

Implementing an advising algorithm in which multiple entries and exits were possible would involve placing traps at each entry and exit and calling the function ADVISE at that point. This could be done, because one can only "GO" to a labelled statement, and PROG labels are easily distinguishable from LISP forms that are to be executed. This has not been done because it has not, as yet, been needed.*

ADVISE

ADVISE, as currently implemented, is designed to modify the interface of a procedure which has only one entry and one exit. ADVISE has four arguments: the name of the procedure, the names of its arguments, the values of its arguments, and the

* There would be some slight complications because of the particular implementation of LISP at Project MAC, where PILOT is now operating. "GO" statements cannot be used when the label is not local. Thus if we inserted a call to ADVISE at each label, and then, inside of ADVISE, wished to execute (GO label), we could not do so. The alternative would be to build our own version of the LISP interpreter inside of the ADVISE function. This would be cumbersome and inefficient.

S-expression definition of the procedure. ADVISE records on the HISTORY list that this procedure has been entered with certain arguments, and retrieves the advice associated with entry to this procedure under the property BEFORE on the property list of the name of the procedure. (We can think of the procedure as having a canonical entry point labelled BEFORE.)

If a LISP form appears under the property BEFORE, instead of a list of advice, ADVISE treats it as a function of one variable and applies it to the single argument HISTORY.* The value of this computation is then used as the advice associated with the entry to the procedure. In this way, the user can achieve the effect of a multiple entry, i.e., different advice can be used for different entering conditions.

Each piece of advice is a LISP computation. ADVISE evaluates in turn each individual piece of advice, making available all information that is available to the original procedure. The evaluation of advice may cause these variables to be modified, or even create new, available variables by modifying the HISTORY list. (Communication between pieces of advice can be achieved this way.) When all of the advice has been evaluated, the procedure itself is executed, and its value is stored on the variable VALUE and put on the HISTORY list.

ADVISE then gets the advice associated with the exit from the procedure from the property AFTER, and operates in a manner similar to that with BEFORE. When all of the AFTER advice has been evaluated, ADVISE restores the HISTORY list and returns as

* HISTORY contains information relevant to the computation. It is described below on page 38.

the value of the procedure the value of the variable VALUE (which may have been changed during the execution of the AFTER advice).

This discussion presumes that the value of each piece of advice in NIL. The user can affect the flow of control - from advice to procedure to advice - by returning a non-NIL value from a piece of advice. If the value is a list, the first element of this list is taken as the value of the procedure, and the rest of the advice is ignored. If this happens BEFORE the procedure is entered, ADVISE binds the first element of the list to VALUE on the HISTORY list, gets the AFTER advice, and proceeds from there. If it happens AFTER evaluating the procedure, the first element of the list is taken as the value of the procedure and returned immediately. In this way, the user can indicate that the original procedure is to be bypassed entirely.

If the value of a piece of advice is an atom other than NIL, it is interpreted by ADVISE as a GO instruction. ADVISE treats the value as a label, and searches for the label in the list of advice, and continues with the evaluation of advice from that point. For example, the user can abandon evaluation of advice without bypassing the original procedure by returning BOTTOM as the value of a piece of BEFORE advice. (TOP and BOTTOM are labels interpreted specially by ADVISE.)

Since ADVICE is a variable made available by ADVISE, the execution of any piece of advice can also modify the advice list. For example, the advice (PROG2 (SETQ ADVICE NIL) NIL) will produce the same effect as the advice (QUOTE BOTTOM), i.e., cause the rest of the advice to be ignored. Similarly the user could interpret GO instructions himself by searching for labels and

modifying ADVICE accordingly.

With the discussion of one more feature, the description of the operation of ADVISE will be complete. This is the provision for modifying classes of procedures. This is done by referring to the property list of the atom ALL, under the properties BEFORE and AFTER as discussed above, before getting the advice specific to this procedure. Since arbitrary LISP functions can appear on these properties, it is clear that one can specify advice for any recursive set of functions. For example, to determine whether or not the procedure in question has called itself more than twice, one need merely search the HISTORY list.

The flow chart in fig. 1 illustrates the operation of ADVISE.

Advising

Advising a function consists of storing a piece of advice on the property list of the function under the appropriate property. If this is the first time the function has been advised, it is also necessary to replace the function definition with a call to ADVISE. Both of these operations may be performed by calling the function provided for this purpose: SYSTEM1.

SYSTEM1 is a function of three arguments: NAME, the name of the function to be advised, ADVICE, the piece of advice, and WHERE, the place (property) it is to be stored. SYSTEM1 appends ADVICE to the list of advice (if any) that appears under the property WHERE. If this is the first time NAME has been advised - as indicated by the fact that the property ADVISED does not appear on NAME's property - SYSTEM1 also replaces the definition

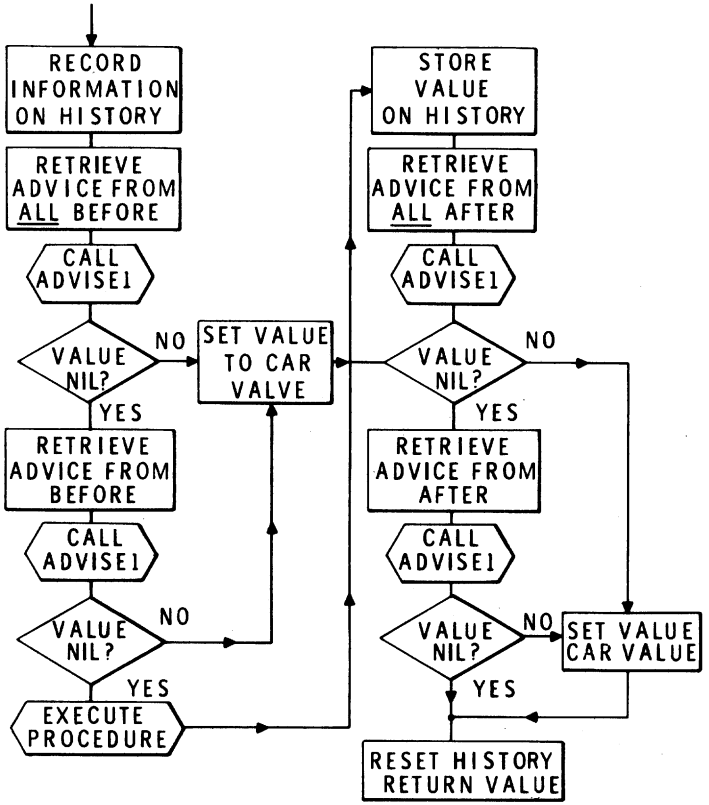


FIG. 1a ADVISE

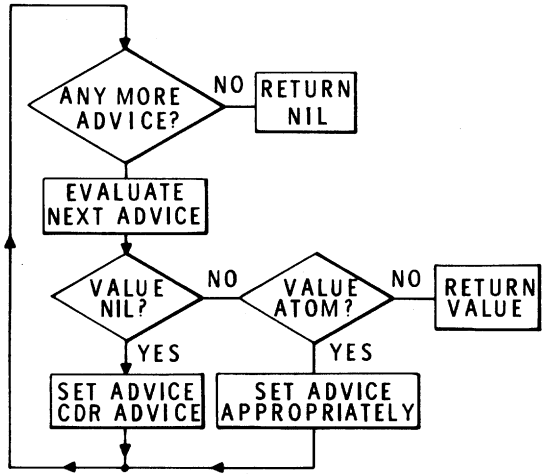


FIG. 1b ADVISE1

of NAME by a call to ADVISE. If NAME is not compiled, SYSTEM1 can get the names of its arguments from its definition (which is also on its property list). If NAME is compiled, SYSTEM1 requests the names of its arguments from the user. SYSTEM1 then redefines NAME, but saves its old definition as the definition of a new function, whose name is placed under the property REALNAME. SYSTEM1 also puts the property ADVISED with value *T* on NAME's property list to indicate that NAME is ready for advising.

Thus if FOO has the definition (LAMBDA (X Y) α), and the user calls SYSTEM1 with NAME = FOO, ADVICE = β, WHERE = BEFORE, the property list of FOO after SYSTEM1 has been executed is:

EXPR (LAMBDA (X Y) (ADVISE (QUOTE FOO) α (QUOTE (X Y))
(LIST X Y)))
BEFORE β
ADVISED *T*
REALNAME OLDFOO

and the property list of OLDFOO:

EXPR (LAMBDA (X Y) α)

If the user wishes to perform other operations with advice, for example, placing the advice at the beginning of the advice list under WHERE, instead of appending it at the end, it is a simple matter to define a function to do this. (In PILOT, the function SYSTEM3 performs this task.) Similarly, by calling EDIT he can specify arbitrary manipulations of advice.

No provision is made specifically for advising procedures that are not LISP functions, even when they satisfy the one entry, one exit requirement. Whereas the definition of a function

can always be found on its property list, locating an arbitrary procedure must be done by prescribing both the name of the function in which it appears and some indication of where in its definition it is. However, it is easy to write a function which uses the editor to locate an arbitrary procedure inside a function and replace the procedure with a call to ADVISE. A similar function already exists for locating and defining as a new function an arbitrary piece of advice, so that one may subsequently prescribe advice on it. This is described under the NAME feature in appendix 2 (page 174).

HISTORY

The HISTORY list is a globally available variable which contains information regarding computation in progress. HISTORY is maintained by the function ADVISE and consequently only functions that have been advised will have their passage recorded on it. The presence of HISTORY means that user programs, or user advice (which is really the same thing), can "look back" and see "what is happening above." This is valuable for avoiding looping, and in making decisions about allocation of resources.

HISTORY has the form of an Alist; that is, it is a list of dotted pairs which represent variable bindings. Thus, it can be used to evaluate a variable, or it can be searched directly by the user's program.

The individual function calls are clearly segmented on HISTORY. This is done by having each call prefaced by an appearance of a special variable named *FN*, followed by a binding for the name of the function. After the name of the function, the arguments of the function are strung out, eventually followed

by the next binding for the variable *FN*. Thus, the segment of HISTORY corresponding to the function FOO, with arguments X and Y, looks like:

(..... (*FN* . ?) (FOO . ?) (X . ?) (Y . ?) (*FN* . ?) ....)

In this segment, the value of *FN* is a pointer to the next entry on the HISTORY list, i.e., the list beginning with (FOO . ?). The value of FOO itself is a dotted pair consisting of a pointer to the next entry on the HISTORY list, i.e., (X . ?) ..., and a pointer to the next (earlier) function call, i.e., (*FN* . ?). The value of X and Y are, of course, whatever their value is.

The structure of the FOO segment of the HISTORY list thus looks like:

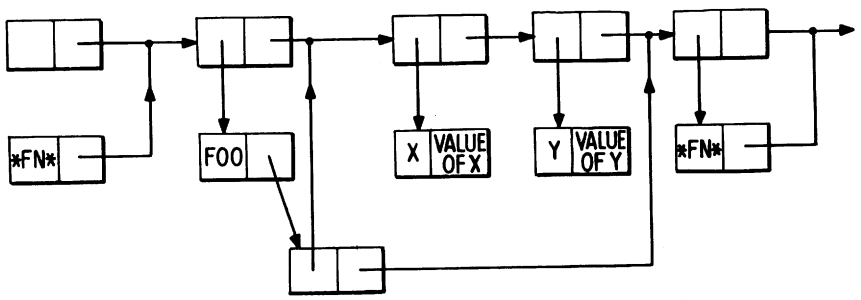


FIG. 2 STRUCTURE OF HISTORY

Because of this structure, one can immediately locate the function called just before FOO - by evaluating (CAADDR FOO).* Similarly, one can locate the last call to the function FIE that occurred before FOO was entered by evaluating FIE against the

* CDR of the value of FOO is the HISTORY list beginning with the last function call. The second pair in this list, CADR, corresponds to the binding of the function name. CAR of this pair is the name itself. Hence CAADDR.

HISTORY list before FOO, i.e., (EVAL (QUOTE FIE) (CDR FOO)), etc.

The HISTORY list can be used to create new variables. In fact, ADVISE does this each time a function is evaluated when it creates the variable VALUE and binds it to the value of the function. This dotted pair, (VALUE . value), is inserted between the binding of *FN* and the binding of the function name. Thus, later functions can determine whether this function is in the BEFORE or AFTER phase, and if AFTER, what the value of the function was.*

The User-PILOT Interface

If we consider the entire system consisting of the user, PILOT, and the user's programs as one program, then it should be possible to modify the interfaces between the user and PILOT, and PILOT and the user's program with the same techniques one uses to modify the interfaces between procedures inside of the user's programs. This section describes modifications of this type that have been carried out in the current version of PILOT.

The user-PILOT-program configuration can be illustrated by the following diagram:

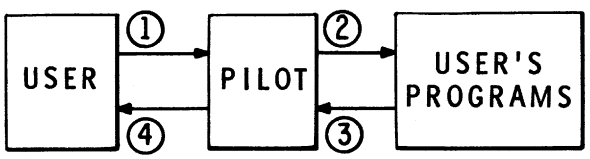


FIG. 3 USER-PILOT INTERFACE

* Since HISTORY records only computations in progress, these bindings last only until the return from the function. Thus VALUE has a binding only during the time that AFTER advice is being evaluated.

In this diagram, the user, (1), requests PILOT to perform an operation, such as advising a user function. PILOT performs this operation on the user's program, (2), and acknowledges completion of the request, (4). When the user's programs are executed, they may interact with PILOT, at (2) and (3), either through the medium of the function ADVISE, or by specifically calling for services provided by PILOT, such as BREAK or FLIP.

It is important to observe that if the user utilizes PILOT in writing and debugging his programs, as well as in modifying them, i.e., if all of his communication with the machine is under PILOT's auspices and go through interface (1), then there is certain amount of tradeoff between efforts at improving interface (1), and those concentrated on interfaces (2) and (3).

For example, we can relax the conventions imposed on communication between the user's programs and PILOT, so that when writing his programs, the user need not be concerned about the details of the interaction at (3). Alternatively, we can impose very stringent requirements on this interaction, but still relieve the user of the burden of conforming to these conventions by transforming his requests into a form which adheres to these conventions at interface (1). The only important features of the process are the two endpoints: relaxed and flexible inputs by the user, and, ultimately, instructions recognized by the machine, i.e., LISP computations. The choice of where along the way the interpretations and transformations take place is arbitrary except for questions of efficiency.

What has been done in the current PILOT system is to implement a collection of powerful text-manipulating functions

within LISP in the form of FLIP, format list processing language. [48] The presence of FLIP makes it easy to introduce, at interface (1), a translation scheme that transforms the user's requests into calls to appropriate LISP functions. This is because FLIP is sufficiently sophisticated to allow a single rule to specify many variations on a particular transformation. This is a necessity for constructing the many-one mapping required for flexible input. A less sophisticated language would either restrict the user excessively or else force him to specify so many different transformations as to be impractical.

In effect, by using FLIP, the user can devise his own conventions and rules, essentially develop his own language, for communicating with PILOT. The interpretation of this source language in terms of LISP functions provide the semantics of the language, which can be expanded by defining new LISP functions as needed, such as SYSTEM3. These correspond to the operations PILOT performs at interface (2). The syntax of the language is also controlled by the user and is therefore easily expanded and modified to suit the user's own ideas as to what is intuitive and natural. (The particular conventions and translation scheme I have adopted for working with PILOT are described in the Appendix 2.)

FLIP is also available to the user for more conventional tasks.* A surprising number of the operations performed by programs fall under the heading of pattern-driven data manipulation. The availability of FLIP considerably simplifies the problem of specifying these operations. The user does not have

* It is the presence of FLIP that makes possible the sophisticated editing available in the PILOT system.

to program each operation anew, nor is he faced with the problem of devising a scheme which will translate and/or interpret these operations at interface (1). All of this may be postponed until FLIP itself is called from within the user's programs. In this case, FLIP may be thought of as improving interface (3), as well as interface (1). Furthermore, enough attention has been devoted to efficiency in the construction of FLIP that the most sophisticated programmer need not hesitate to use it in writing his own programs.

Although most of the effort at modifying the user-PILOT interface has been directed at interface (1), the user may also wish to improve interface (4). The appendix describes some modifications I have carried out with advice that affect this interface, especially with regards to the procedure followed when an error occurs somewhere between the user's initial request and the successful completion of the indicated operation at (2).

This empty page was substituted for a blank page in the original document.

CHAPTER 4

FACILITIES IN THE PILOT SYSTEM

This chapter describes three of the facilities provided for the user by the PILOT system. Central to these is the language FLIP which is used by the system to process the user's requests, as well as being available to the user for a variety of tasks. FLIP is integral to EDIT, a collection of fairly sophisticated editing routines that may be readily expanded by means of advice. BREAK and BREAKPROG provide facilities for arresting the flow of computation at a procedural interface so that the user can perform computations, perhaps make modifications in the system, and then either continue with the computation or specify some alternate path.

I. FLIP

FLIP incorporates a notation and a programming language for expressing, from within the LISP system, string transformations such as those performed in COMIT or SNOBOL. These transformations may be exemplified by the following instructions for a transformation: find in this string the substring consisting of the three elements immediately preceding the first occurrence of an a, and find the element just after the occurrence of a b which follows this a; if such elements exist, exchange the position of the three elements and the one element, delete the a, and replace the b by a c.

Transformations of this type are fundamental for editing, translating, in fact for performing any operation that is basically pattern-driven, i.e., specified by giving the form of the output in terms of the form of the input. However, they are difficult to express in the explicit function-oriented nature

of LISP, although each could be individually programmed. A notation for expressing such transformations is the basis for a number of programming languages that exist today, such as COMIT,[51] SNOBOL,[15] and AXLE.[9] Each provides a formal method for selecting substrings from a string, and then indicating the structure of the transformed string.

These formalisms make it easy to write rules which perform string transformations such as rearrangement, deletion, insertion, and selection of elements from contents. However, it is cumbersome to express in these languages some of the operations which are expressed quite easily in LISP. Some of the latter operations depend very strongly on the fact that LISP can have sublists within lists to unlimited depth, whereas COMIT has lists only to depth 3 and SNOBOL and AXLE deal only with linear strings.

An obvious solution to this notational difficulty is to provide both types of language capability, function-directed and format-directed list processing notation, within the same programming system. These two capabilities are provided in PILOT by embedding FLIP[7,49] in the LISP 1.5 programming system.*

FLIP Transformations

A transformation is specified in FLIP by providing a pattern, which must match the structure to be transformed, and a format,

*The implementation of FLIP in LISP 1.5 is based upon but is a considerable generalization over, programs and writings of Bobrow.[5,6] In addition, it has been influenced by features of the string processing languages described above, as well as by those of CONVERT[19] another string processing language embedded in LISP.

which specifies how to construct a new structure according to the segmentation, or parsing, specified by the pattern. These patterns and formats are greatly generalized versions of the left-half and right-half rules of COMIT and SNOBOL. For example, elementary patterns and formats can be variable names, results of computations, disjunctive sets, or repeating subpatterns; predicates can be associated with elementary patterns which check relationships among separated elements of the match; it is no longer necessary to restrict the operations to linear strings since elementary patterns can themselves match structures. Furthermore, it is relatively easy to expand the semantics of FLIP, adding new types of patterns and formats, by defining appropriate LISP functions.

Since FLIP is embedded within LISP, it does not have its own control mechanisms. In fact, several different useful executive programs have been written in LISP to facilitate using sets of rules. Some of these do the following:

1. Repeat use of each rule until it fails, and then go on to the next.

2. Every time a rule is successful go back to the top of the set of rules. On failure go to the next rule.

3. On a successful match, go to a specified labelled rule (similar to COMIT).

(The latter algorithm is embodied in the LISP function TRANSFORM, which is used in the translating scheme as well as in the editor.) Since executive programs can easily be changed or written anew for each applications, the flow of control between rules is obviously not an important factor in the design of FLIP.

Notation in FLIP

Let us return to the transformation described earlier: find in this string the substring consisting of the three elements immediately preceding the first occurrence of an a, and find the element just after the occurrence of a b which follows this a; if such elements exist, exchange the position of the three elements and the one element, delete the a, and replace the b by a c.

In COMIT, this operation is expressed by the following rule:

```
$ + $3 + a + $ + b + $1 + $ = 1 + 6 + 4 + c + 2 + 7
```

In the COMIT notation, the "$" matches anything, the $n where n is a number, matches a segment of length n, and x matches x, i.e., a segment of length 1 consisting of the single item x. The numbers in the right hand side of the rule refer to the corresponding elements in the left hand side of the rule, e.g., "5" refers to the single element "b", "2" refers to the three elements preceding "a."

The external notation of FLIP is in fact quite similar to that of COMIT. Giving FLIP the pattern ($ $3 A $ B $1 $) and the format (1 6 4 C 2 7) will cause the transformation described above to be performed, i.e., (FLIP X (QUOTE ($ $3 A $ B $1 $)) (QUOTE (1 6 4 C 2 7))) specifies a LISP computation which transforms the variable x according to this rule.

However, since FLIP may also be used on nonlinear strings and has in it many features which do not have counterparts in COMIT, e.g., use of predicates, repeating subpatterns, etc., it has been necessary to expand the COMIT notation considerably. For example, to cancel out the common factor in LISP expressions

such as (QUOTIENT X (TIMES A B X Y)), one uses the pattern (QUOTIENT $1 (TIMES $ (/T 2) $)) and the format (TIMES (/T 3 2) (/T 3 4)). The "/T" indicates that the numbering should begin at the "top" of the parsing, i.e., (/T 3 2) is the second element, in the third element of the parsing, or (A B), and (/T 2) is the second element, or X. (Alternatively, one can specify that the numbering is to begin at the current level, or up a certain number of levels.) To find a string of three elements which are immediately followed by their mirror image, one uses the pattern ($ $3 $3 / (EQUAL (= REVERSE 2)) $), where the predicate (EQUAL (= REVERSE 2)), associated with the second "$3", signifies that the result of applying the LISP function REVERSE to the element corresponding to the first $3, indicated by "2", must be equal to this element in order for the pattern to match.

However, it is not the intent of this discussion to describe the operation of FLIP in detail, but instead to indicate the ways in which it can be useful, and the problems to which it is applicable. For this purpose, the manner in which certain operations are expressed is not at all important, especially since the current notation is arbitrary and ad hoc. (A large part of the awkwardness of this notation is due to the clumsy way in which reading and printing occur in the present LISP 1.5 system, and to the dearth of available symbols.) In any event, since utilization of FLIP involves a translation from an external language to a more efficient form for internal use, it would be possible with more sophisticated translators to provide whatever notation the user wishes. Thus the important thing about FLIP is the semantic features made available by it. The examples on the next pages are designed to illustrate some of these.

Applications of FLIP

Translation. FLIP was originally conceived and implemented for a specific purpose: to provide in PILOT a capability for transforming user requests into LISP computations. The details of this translating scheme are described in appendix 2. In this example, I shall motivate and construct in greater detail one of these transformation rules.

Prefix notation plays an important part of the LISP formalism. Relations are expressed with the name of the relation first, e.g., (X IS A MEMBER OF Y) is (MEMBER X Y), (X IS LESS THAN Y) is (LESSP X Y). This is convenient because it puts the name of the operation in a canonical position, and so avoids the problem of identifying that member is the key word in (X IS A MEMBER OF Y). However, since English is basically an infix notational language, the user must continually perform mental transformations when programming in LISP. The translation scheme implemented in PILOT is designed to lessen the user's burden.

This translator is basically a sequence of FLIP rules which perform transformations on the input. Thus including a rule such as:

```
((-- $1 IS A MEMBER OF $1 --) (-- (MEMBER 2 -2) --))
```

will allow the user to write

```
(...X IS A MEMBER OF Y ...)
```

for

(... (MEMBER X Y) ....).*

Similarly, ((-- $1 IS LESS THAN $1 --) (-- (LESSP 2 -2) --)), and ((-- $1 IS GREATER THAN $1 --) (-- (GREATERP 2 -2) --)), etc. However, as we introduce further "IS RULES," the translation process will be slowed down considerably, because of the increasing number of attempted matches. In addition, the question of space may become crucial. We would like to have a single rule handle all of the "IS" transformations. This would have the added advantage that we could also transform (X IS NOT A MEMBER OF Y) into (NULL (MEMBER X Y)) with the same rule.

The way to construct such a rule is to use the disjunctive "EITHER" pattern, with a variable pattern name. For example, if we store the patterns for the IS transformations on the property list of the atom IS under the property PATTERNS, then (-- $1 IS (EITHER (= GET IS PATTERNS)) --) will match if the input list is of the form of one of the IS PATTERNS. The format (-- ((EITHER (= GET IS FORMATS))) --) will then perform the desired transformation, selecting the format corresponding to this match. To incorporate the NOT feature, we write instead

---

* A word about notation:
1. The above rule translates the same as (($ $1 IS A MEMBER OF $1 $) (1 (MEMBER 2 -2) -1)). "--" means either "$" or "1" or "-1" or even "NIL," depending on the context. It represents a DWIM statement - "DO WHAT I MEAN."
2. Negative numbers, such as -2, -1, serve the same function as positive numbers, they refer to elements in the parsing. However, with negative numbers, the numbering starts from the right hand side and counts backward. Thus in the above, (MEMBER 2 -2) is the same as (MEMBER 2 7) and also (MEMBER -7 7).

```
(-- $1 IS (EITHER (NOT) --) (EITHER (= GET IS PATTERNS)) --)
```

and

```
(-- ((EITHER (NULL (EITHER (/T -2) (= GET IS FORMATS)))
((EITHER (/T -2) (= GET IS FORMATS))) )) --)
```

With this rule, it is necessary to specify by (/T -2) which of the two "EITHER's" we are referring to in the match (FLIP will select the first one if none is specified).

Now, if we put on IS PATTERNS: (MEMBER OF $1), GREATER THAN $1), (LESS THAN $1), even (ATOMIC) and (A NUMBER); and on IS FORMATS: (MEMBER 2 (/T -2 -1)), (GREATERP 2 (/T -2 -1)), (LESSP 2 (/T -2 -1)), (ATOM 2), (NUMBERP 2), etc., the appropriate transformations will be performed. Furthermore, it is a simple matter to write another rule which automatically does this, i.e., we can say (X IS GREATER THAN Y MEANS GREATERP) and (GREATER THAN $1) will be put on IS PATTERNS and (GREATERP 2 (/T -2 -1)) on IS FORMATS; if we say (X IS A NUMBER MEANS NUMBERP), then (A NUMBER) will go on IS PATTERNS and (NUMBERP 2) on IS FORMATS, etc.

Output

Frequently a programmer will content himself with relatively sterile output because the extra labor involved in programming fancier output does not justify the returns. FLIP makes it easy for programs to communicate with the user in text, as opposed to list structure.

Suppose a program deals with data such as (AT PENCIL Y), (NIL AT PENCIL COUNTY), and (((AT PENCIL DESK) (AT DESK HOME)) AT PENCIL COUNTY). The first list is the representation of the

question: "Is the pencil at any y?" The second that of the statement, "The pencil is at the county" and the third, "If the pencil is at the desk and the desk is at the home, the pencil is at the county."* We would like to have the program output this information this latter way instead of the way it is internally represented.

To do this, we proceed as follows:

First we define a function PHRASE which transforms (AT PENCIL COUNTY) into (THE PENCIL IS AT THE COUNTY). The definition of this function, using FLIP, is:

```
(LAMBDA (X) (FLIPQ X
($1 (EITHER ($1 / (VARIABLE)) ($1)) (EITHER ($1 / (VARIABLE))
     ($1)) )
((EITHER (ANY) (THE)) 2 IS 1 (EITHER 3 (ANY) (THE)) 3) ))
```

FLIPQ is the same as FLIP, except we don't have to quote the latter two arguments. VARIABLE is a function which is true if its argument is a variable, false otherwise. Note that PHRASE applied to (AT PENCIL Y) is (THE PENCIL IS AT ANY Y).

Next we define a function QUESTION which transforms (AT PENCIL COUNTY) into (IS THE PENCIL AT THE COUNTY). The definition of this function is:

```
(LAMBDA (X) (FLIPQ X
($1 (EITHER ($1 / (VARIABLE)) ($1)) (EITHER ($1 / (VARIABLE))
     ($1)) )
(IS (EITHER (ANY) (THE)) 2 1 (EITHER 3 (ANY) (THE)) 3) ))
```

* This example is from Chapter 5, Experiments with a Question-Answering System.

Using these two functions, we can define OUTPUT to handle the three different data types:

```
(LAMBDA (X) (FLIPQ X
(EITHER (NIL --) ($1 / (ATOM) --)
(($1 (EITHER ($1) --)) --) )
(EITHER
((** (= PHRASE 2)) (= PERIOD))
((** (= QUESTION (/T 1))) (= QMARK))
(IF (** (= PHRASE (= CAR (/C 1 1))))
(EITHER (AND (** (= PHRASE 1))) --)  (= COMMA)
(** (= PHRASE 2)) (= PERIOD) ) ) ))
```

In this definition, "**" denotes that the result of the LISP computation is not to be treated as a single element and inserted, but to be treated as a list and appended, so that the resulting structure will be a linear list. The (/T 1) in the call to QUESTION gives it as input what matched the top level EITHER, or the entire list X. Finally, the (/C 1 1) denotes the first element in the first element of the current structure. In this case, it is the same as writing (/T 1 1 1), or (= CAR 1).

Searching and Sorting

A surprising number of the more common tasks performed by programs fall under the heading of pattern-driven data manipulation. Thus, they could be written in FLIP. For example, when we search a list for a particular item or group of items, as specified by some relationship, we are performing the same operation as that performed by the function MATCH in FLIP. Even such mundane operations as sorting a list can be expressed simply in the FLIP notation.

Suppose we wish to define a function which will take the list (X X Y W X Z Y) and produce as output the list ((3 X) (2 Y) (1 W) (1 Z)). The following function, using TRANSFORM, (see page 47), will do this:

```
(LAMBDA (X) (TRANSFORM X (QUOTE (
LOOP ((-- $1 / (ATOM)    (REPEAT $ (/T 2)) --)
     (-- 4  (REPEAT N 2)  ((= (CAR N)) 2)) LOOP)  ))))
```

In this rule, N is the index of repetition. When the REPEAT format completes operation, CAR of N is the number of times it repeated.

If instead we wanted as output (X X X Y Y W Z), we would add to the above list of rules for TRANSFORM:

```
((REPEAT $1)  (REPEAT (= CAR 1)  (= CADR 1)))
```

This rule will transform ((3 X) (2 Y) (1 W) (1 Z)) into (X X X Y Y W Z).

* * *

II. EDIT

Programmers must have a way of editing their programs. This is a simple consequence of the fact that programmers make mistakes. Unfortunately, however, editing facilities are often primitive; the limiting factor in debugging programs may be the interaction time with a keypunch.

One approach to finessing the duplicate button on the key punch is to construct a context editor for the source material,

usually paper tape or card decks. This is the approach of Tape Editor[42] and ED.[11] Here the user moves an imaginary pointer through his program listing using context search, e.g., locate the character string "CONS (CAAR X," and performs insertions, deletions, and replacements. The editing program makes the corresponding changes in the source material and issues the user a fresh version at the end of the editing session.

For LISP programmers, the above procedure necessitates leaving the LISP system and the original program. This may be undesirable if the user is in the midst of a debugging sequence, especially if returning to the LISP system involves a lengthy loading process. Another approach to the problem of editing, therefore, is to provide some form of editing facility within the LISP programming system. This is the approach of Martin[28] and Bobrow.[8] Here the user has the added advantage that he can edit list structure, instead of text, although this may make it difficult to correct a simple parenthesis error. The operations corresponding to moving the pointer allow the user to refer to pieces of list structure. Similarly, insertion, deletion, and replacement commands specify changes in the structure. At the end of the session, the editor produces a new version of the list structure. The programmer can then proceed with his debugging immediately.

From the standpoint of the LISP user, this latter approach is superior. However, for efficient LISP editing, the properties of a structure editor and a text editor are both required. The user should be able to manipulate individual parentheses as easily as pieces of list structure.

Another desirable feature of an editing program is a language for expressing editing operations. The absence of a language tends to preclude conditional operations. The user cannot specify operations involving decisions, even simple ones, such as find an "X" that immediately follows a "Y" - except by searching for a "Y" and examining the next element himself.* A language is also necessary to enable the user to define new operations, without reprogramming the editor.

Using FLIP for Editing

The presence of FLIP provides a language for describing editing operations. In fact, all that is necessary to construct an editor is to write an executive program which accepts requests from the user and calls FLIP. The insertions, deletions, and replacements of editing are specified by FLIP patterns and formats. Furthermore, since FLIP rules are themselves list structure, it is easy to modify them using other FLIP rules - e.g., by giving advice. In this way, a sophisticated editor can be built around the FLIP language with very little additional effort.

Such an editor has been included in the PILOT system. The following discussion presents its salient features.[48]

Example

To give the general flavor of editing using FLIP, suppose the definition of the function FOO is

* TECO,[44] tape editor and corrector, is an excellent example of the advantages of an editor with a language.

```
(LAMBDA (X) (PROG NIL (COND
((EQ (CAR X) -1) (RETURN NIL)))
(SETQ Y (PLUS (Y CAR X)))
(SETQ X (CDR X))
(GO START)  ))
```

Let us add Y to the argument list, and label the COND statement START.

```
edit (foo expr nil)
 (match -- x --)*          [find the left-most x]
 (form 1 2 y 3)            [follow it by y]
 (match -- nil --)         [the first NIL]
 (form 1 2 start 3)
 stop
  FOO                      [value of EDIT]
```

We could perform both changes in a single match and construct if we desired. Also, we could check our intermediate results by examining the output of the matches.

```
edit (foo expr nil)
(flip (-- x -- nil --) (1 2 y 3 4 start 5))
(match -- prog -- cond --)
1                          [what is 1 in last match?]
     (LAMBDA (X Y)  (
3
     NIL START (           [this is 3]
stop
     FOO                   [the value of EDIT]
```

Flattening Lists and Balancing Parentheses

For all intents and purposes, in the above example, we were editing a string of atoms. This effect is achieved by "flattening" all S-expressions that are to be edited into a single list of atoms, substituting the special atoms L* for left parentheses,

---

* "form" is used instead of "cons" for "construct" because the word "cons" would be confused with the LISP function CONS.

R* for right parentheses, and P* for dot. For example, ((A . B) (C . D)) flattens to (L* L* A P* B R* L* C P* D R* R*). Since L*, R*, and P* are atoms the same as X, Y, and NIL, we can insert and delete them as well as any other. While inside of the editor, we can even manipulate "partial" lists such as "(LAMBDA (X Y) (", represented as (L* LAMBDA L* X Y R* L*). The only restriction is that the list must "unflatten" correctly when we wish to leave the editor.

To restore the properties of list structure to the editor, i.e., to allow us to refer to pieces of list structure as well as strings of atoms, we now expand the semantics of FLIP by adding a new elementary pattern, UPN. This elementary pattern signals FLIP to find the nth matching pair of parentheses, that is, to go UP n pairs of parentheses starting from the current position. In effect, what the UPN pattern says, for n=2, is "I didn't really want to match with (..) but with the list containing the list containing (..). However, it was easier to find this list by first locating (..), and then backing up two sets of parentheses." Thus in the example on the previous page, we could find the structure ((EQ (CAR X) -1) (RETURN NIL)) by matching with (-- ((EQ (CAR X) -1) (RETURN NIL)) --) (FLIP will automatically flatten the input pattern), or by matching with (-- EQ UP2 --), or (-- CAR UP3 --), or (-- RETURN UP2 --). The UPN pattern would then match with the structure ((EQ (CAR X) -1)(RETURN NIL)), which could be transformed as desired.

Adding New Operations

Suppose we want to

INSERT (SETQ X 1) (SETQ Y NIL) AFTER CAR -- CDR UP2

i.e., after the UP2. We match with (-- CAR -- CDR UP2 --), and construct with (1 2 3 4 5 (SETQ X 1) (SETQ Y NIL) -1). To

REPLACE CONS UP2 WITH (LIST Z),

we match with (-- CONS UP2 --) and construct with (1 2 (LIST Z) -1).

In fact, to insert α after β, match with (--β--) and construct with (1 2 3 ... n -2 α -1), where n is the length of β. To insert α before β, match with (-- β --) and construct with (1 2 3 ... n α -2 -1). To replace β with α, match with (-- β --) and construct with (1 2 3 ... n α-1), etc.

This suggests that it should be possible to give the editor requests such as (INSERT (SETQ X 1) (SETQ Y NIL) AFTER CAR -- CDR UP2), and (REPLACE CONS UP2 WITH (LIST Z)), by defining the operations INSERT AFTER, INSERT BEFORE, REPLACE WITH, etc.

This is in fact easy to accomplish by first transforming the request to EDIT according to a set of EDIT RULES. Adding new FLIP rules to this list allows the user to define new operations. For example, to define

(REPLACE ... WITH ...) we add the rule

```
((REPLACE $ WITH $)  (FLIP ($ 2 $) ((REPEAT (= LENGTH 2)
                     (= (CAR N))) 4 (QUOTE -1)))).
```

This transforms (REPLACE CONS UP2 WITH (LIST Z)) into (FLIP ($ CONS UP2 $) (1 2 (LIST Z) -1)), which is then recognized as a request for FLIP. New operations can even be defined in terms of old ones, e.g., ((DELETE $) (REPLACE 2 WITH)) allows the user to specify (DELETE CONS UP2). With a little practice, the user can define fairly complicated operations such as (CHANGE ALL α TO β), (SUBEXP α BEFORE β) (which allows one to move structure from one place to another) and (WHAT IS α), for interrogating the current status of the edited structure. In this way, the user can build up his own vocabulary and language for editing, always returning to the basic FLIP operation for complicated and/or special purpose operations.

* * *

III. BREAK AND BREAKPROG

In order to edit (or advise) an incorrect procedure, we must first know what procedure is at fault, and the precise nature of the problem. Finding this out can be a very difficult task. If the error is such that the program does not produce any meaningful output at all, there may be no course of action left but to examine the action of a large number of procedures in detail. Even when we have some idea of where the trouble spot may lie, and in inkling of what it is, we must still be able to examine closely the operation of a procedure. We want to find out what changes it makes, if any, in its essential

variables. Basically, what we want to do is arrest the flow of computation at the entry and exit to a procedure, perform various computations, and then either continue with the normal flow of control, or indicate alternate routing.

There is great similarity between this operation and that of advising. In fact, the two are identical, except that with advising, the computations are prespecified on the property list of the procedure, whereas with this operation, which I call breaking, they are entered through the keyboard at the time of the break. This of course is the essential point of breaking. Since the user does not know what the trouble is, he cannot fully anticipate the questions (computations) he will want to ask, prior to the time of the break. In general, each question will depend on the "answer" he receives to the previous one.

Breaking is implemented in PILOT by a function BREAK1.[48] BREAK1 takes as input the definition of a procedure, and allows the user to execute LISP computations before and after evaluating this procedure. These computations are entered from the keyboard, and, after execution, their value is printed.

BREAK1 plays a role in breaking similar to that of ADVISE in advising. However, since efficiency is not important in BREAK1,* the various advising conventions concerning exit from a procedure have been replaced with four special commands: QUIT, STOP, RETURN, AND EVAL, for which BREAK1 makes a special check.

* In general, whenever input or output is required, efficiency of computation is not important, because the computation time is so small compared with the time required for the input and output processes.

QUIT, STOP, AND RETURN specify exits from the entire breaking operation: QUIT induces a LISP error; STOP is the normal (unbroken) return from the procedure; and RETURN specifies a return with some other value, i.e., via another computation. EVAL is used when the user wishes to evaluate the procedure, without exiting from the break. This corresponds to going from the BEFORE to AFTER phase in advising, except that with breaking, this can be done more than once. For example, after an EVAL command, the user can check the value of the procedure, make some changes, and EVAL again.

Breaking a procedure involves replacing its definition with a call to BREAK1. Again, note the similarity to advising. There are two functions available for this purpose. BREAK is used when the procedure is a LISP function. BREAKPROG is used when the procedure occurs inside of a LISP function. BREAK gets the function definition from the property list. BREAKPROG calls EDIT to locate the procedure in question and to make the appropriate changes. Since one of the arguments of BREAK1 is a breaking condition, the user can specify that a break is to be conditional upon the result of some computation, and thereby postpone examination of the procedure until a crucial point in the calculation occurs.

This empty page was substituted for a blank page in the original document.

CHAPTER 5

EXPERIMENTS WITH A QUESTION-ANSWERING SYSTEM

Two detailed examples are presented in this chapter and the next. They illustrate the use of the PILOT system. An attempt has been made to give the reader the flavor of an actual session at the console. The complete user-PILOT dialogue is included, along with anecdotal comments explaining what is happening.

Preface

Because it is impossible to allow to each reader himself interaction with the system, I have tried, in these chapters, to give its flavor by going through an example; I have attempted to impart the idea behind each interaction without dwelling on the details. Appendix 2, Using PILOT, delves more deeply into the conventions and operation of the system.

Experiments with a Deductive Question-Answering System

In 1964, Fischer Black programmed in LISP a Deductive Question-Answering System.[2] This system is similar to the Advice Taker proposed by McCarthy,[31] and solves McCarthy's "airport problem," among others.

In the airport problem, the program is supplied with certain facts: at (I,desk) (which is McCarthy's formalization of "I am at the desk"), at(desk,home),at(car,home),at(home,county), at(airport,county),walkable(home),drivable(county), along with general and specific rules relating those facts, such as the transistivity of the "at" relationship: at(x,y) at(y,z) at(x,z).

It is then asked to solve the "problem" posed by the premise "want(at,(I,airport))", in other words to produce a deductive chain which terminates with "at(I,airport)."

The operation of Black's system can better be explained with a simpler corpus. Let us assume the program is given:

in(pencil,desk),
in(desk,home),
in(home,county),
in(x,y)→at(x,y),
in(x,y),at(y,z)→at(x,z).

When asked the question "Is my pencil at the county," i.e., at(pencil,county), the program looks for a statement whose consequent matches the question, and finds two: in(pencil,county)→ at(pencil,county), and in(pencil,y),at(y,county)→at(pencil,county). It then considers as a subquestion "is my pencil <u>in</u> the county" and finding no statements that match, concludes that this question cannot be answered. It therefore considers as a subquestion, the first antecedent in the remaining statement, namely "in(pencil,y)," which asks "what is my pencil in?" "in(pencil,y)" matches "in(pencil,desk)." Since this is a known fact, the deduction is immediate, and since there are no other matches, the answer to the question "in(pencil,y)" is "in(pencil,desk)." The program then attempts to answer "at(desk,county)," because then it could deduce "at(pencil,county)," etc.

One of the interesting problems of this system is that endless deductions can result because the same question occurs as a subquestion of itself. For example, if the corpus were

at(pencil,desk),at(desk,home),at(home,county),at(x,y) & at(y,z)→ at(x,z), then given the question "at(pencil,county)," the subquestion "at(pencil,y)" would keep repeating. Dr. Black discusses various ways to prevent endless deduction in his thesis, and raises various objections to each of them. Unfortunately, implementing and testing each method involved considerable reprogramming. D.G. Bobrow suggested that PILOT could be used to make these modifications and that this would provide an excellent test for it. In particular, since the procedures used in this example would not only be compiled subroutines but would have been written by an entirely different person, it would demonstrate whether or not PILOT really allowed the user to think of a procedure as a little black box with input-output characteristics. Accordingly, I copied the function definitions for Black's system from the appendix of his thesis, and loaded them into PILOT.

A Summary of the Experiment

Since the only output provided by Black's program was an exhaustive trace of the two main functions, SOLUTION1 and SOLUTION2, the first step was to get the program to print out the deductive chain in some readable fashion. When this was done, I discovered that the program was written to produce all possible answers to a question. When the question contained no variables, e.g., at(pencil,county), this meant that the program would continue to look for an alternative way of answering the question even if it had already satisfactorily deduced the answer.* This situation was readily corrected by advice.

* The way the program operates is as follows: if the question is (AT PENCIL Y), it returns a list of all the facts satisfying the question. If the question is (AT PENCIL COUNTY), it also returns a list of all facts satisfying the question, but in this case there can be only one - (AT PENCIL COUNTY). Essentially what it does is say "Yes, the pencil is at the county."

At this point, I decided that it would be easier for me to follow the deduction if the output were in a more readable format. Dr. Black had described the internal representations used for questions, facts, and deductive rules. I therefore wrote three functions, QUESTION, PHRASE, AND CLAUSE, which used FLIP, and transformed the internal representation into English. PHRASE would take something of the form (AT PENCIL COUNTY) and transform it into (THE PENCIL IS AT THE COUNTY). QUESTION produced (IS THE PENCIL AT THE COUNTY Q) from (AT PENCIL COUNTY), (when it was designated as a question). CLAUSE would take an arbitrary expression, decide whether it was a question or a statement, and then perform the appropriate transformation, e.g., (((AT PENCIL Y) (AT Y COUNTY)) AT PENCIL COUNTY) become (IF THE PENCIL IS IN ANY Y AND ANY Y IS AT THE COUNTY, THE PENCIL IS AT THE COUNTY).

The protocol that follows gives the definition of these functions. It shows how BREAK was useful in debugging them and how advice was used to correct the one bug that was found.

After I had the program more or less talking to me, I attempted to solve the original problem, that of endless deduction. The first solution was to limit the number of questions considered. When this limit was set at 2, the program could not get an answer to "at(pencil,county)." Since I was going to play with this parameter, I advised the function to allow me to input this parameter along with the question. Giving it a limit of 3, the program could answer "at(pencil,county)," although it was clear it did so inefficiently. I also tried the question "at(pencil,y)." With a limit of 1, the program got one answer: at(pencil,desk). With a limit of 2, it got two

answers: at(pencil,home) and at(pencil,desk). With a limit of 3, after much labor, all three answers, at(pencil,county), at(pencil,Home) and at(pencil,desk) were obtained.

The last modification undertaken was to instruct the program to look for a repeated subquestion, and if one were encountered, to return with any answers already found to that question instead of considering it anew. With this modification the program obtained all three answers to at(pencil,y) in a very satisfactory manner, with less than half of the effort of the previous method (i.e., setting an arbitrary limit on the number of recursive function calls allowed).

Protocol

Initially the only facility for monitoring the operation of the program was the trace feature in LISP. I traced SOLUTION1 and SOLUTION2, which were the main functions. From the text of the thesis, I knew that (AT PENCIL COUNTY) was a question, and (NIL IN DESK HOME), (((IN PENCIL COUNTY)) AT PENCIL COUNTY) were the way statements were represented. It thus became clear that SOLUTION1 handled questions and SOLUTION2 took a statement and tried to find a deduction that produced it.

```
solution1 ((at pencil county))

 ARGUMENTS OF SOLUTION1
 (AT PENCIL COUNTY)

 ARGUMENTS OF SOLUTION2
 (((IN PENCIL COUNTY)) AT PENCIL COUNTY)

 ARGUMENTS OF SOLUTION1
 (IN PENCIL COUNTY)
```

```
VALUE OF SOLUTION1
NIL

VALUE OF SOLUTION2
NIL

ARGUMENTS OF SOLUTION2
(((IN PENCIL Y) (AT Y COUNTY)) AT PENCIL COUNTY)

ARGUMENTS OF SOLUTION1
(IN PENCIL Y)

ARGUMENTS OF SOLUTION2
(NIL IN PENCIL DESK)

VALUE OF SOLUTION2
((IN PENCIL DESK))

VALUE OF SOLUTION1
((IN PENCIL DESK))

ARGUMENTS OF SOLUTION2
(((AT DESK COUNTY)) AT PENCIL COUNTY)

ARGUMENTS OF SOLUTION1
(AT DESK COUNTY)

ARGUMENTS OF SOLUTION2
(((IN DESK COUNTY)) AT DESK COUNTY)

ARGUMENTS OF SOLUTION1
(IN DESK COUNTY)

VALUE OF SOLUTION1
NIL

VALUE OF SOLUTION2
NIL

ARGUMENTS OF SOLUTION2
(((IN DESK Y) (AT Y COUNTY)) AT DESK COUNTY)

          etc.

ARGUMENTS OF SOLUTION2
(((IN COUNTY Y) (AT Y COUNTY)) AT COUNTY COUNTY)

ARGUMENTS OF SOLUTION1
(IN COUNTY Y)

VALUE OF SOLUTION1
NIL

VALUE OF SOLUTION2
NIL

VALUE OF SOLUTION1
NIL

VALUE OF SOLUTION2
NIL
```

```
VALUE OF SOLUTION2
NIL

VALUE OF SOLUTION1
((AT HOME COUNTY))

ARGUMENTS OF SOLUTION2
(NIL AT DESK COUNTY)

VALUE OF SOLUTION2
((AT DESK COUNTY))

VALUE OF SOLUTION2
((AT DESK COUNTY))

VALUE OF SOLUTION2
((AT DESK COUNTY))

VALUE OF SOLUTION1
((AT DESK COUNTY))

ARGUMENTS OF SOLUTION2
(NIL AT PENCIL COUNTY)

VALUE OF SOLUTION2
((AT PENCIL COUNTY))

VALUE OF SOLUTION2
((AT PENCIL COUNTY))

VALUE OF SOLUTION2
((AT PENCIL COUNTY))

VALUE OF SOLUTION1
((AT PENCIL COUNTY))
((AT PENCIL COUNTY))


csetq (corpus corpus2)

((NIL AT PENCIL DESK) (NIL AT DESK HOME) (NIL AT HOME
COUNTY) (((AT X Y) (AT Y Z)) AT X Z))
solution1 ((at pencil county))

ARGUMENTS OF SOLUTION1
(AT PENCIL COUNTY)

ARGUMENTS OF SOLUTION2
(((AT PENCIL Y) (AT Y COUNTY)) AT PENCIL COUNTY)

ARGUMENTS OF SOLUTION1
(AT PENCIL Y)

ARGUMENTS OF SOLUTION2
(NIL AT PENCIL DESK)
```

```
VALUE OF SOLUTION2
((AT PENCIL DESK))

ARGUMENTS OF SOLUTION2
(((AT PENCIL U) (AT U Z)) AT PENCIL Z)

ARGUMENTS OF SOLUTION1
(AT PENCIL U)

ARGUMENTS OF SOLUTION2
(NIL AT PENCIL DESK)

VALUE OF SOLUTION2
((AT PENCIL DESK))

ARGUMENTS OF SOLUTION2
(((AT PENCIL Y) (AT Y Z)) AT PENCIL Z)

ARGUMENTS OF SOLUTION1
(AT PENCIL Y)

ARGUMENTS OF SOLUTION2
(NIL AT PENCIL DESK)

VALUE OF SOLUTION2
((AT PENCIL DESK))

ARGUMENTS OF SOLUTION2
(((AT PENCIL U) (AT U Z)) AT PENCIL Z)

ARGUMENTS OF S
```

Here I used corpus2, the corpus in which looping may occur. As can be seen from the trace, the program is in an endless deduction.

What I plan to do is to define a new function, SOLVE, which will call SOLUTION1 and initiate the deduction. Each time I enter SOLUTION1 or SOLUTION2 during the deduction, I will save their arguments on a dummy variable - which I will call SOLUTION. If when I leave the function, its value is NIL, I'll then remove its argument from SOLUTION. SOLUTION is thus a first-in first-out list of arguments for SOLUTION1 and SOLUTION2. When I get finished, SOLUTION will have all of the questions and statements that produced non-null values, i.e., those actually part of the deduction.

```
(define solve (x) as solution1 x)
 (SOLVE)
```

Defining SOLVE.

```
(tell solve before, do bind solution to nil)
 SOLVE
```

Telling SOLVE to BIND solution to NIL. The DO means this advice is to be executed without disrupting the normal flow of control - i.e., SOLUTION1 will still be entered.

```
(tell solution1 before, do save x on solution)
 SOLUTION1
```

Telling SOLUTION1 to SAVE on SOLUTION.

```
(translate ((either (before $1) (after $1) ($1 $1) ($1))
: --) as (tell (either (2 1) (2 1) (1 2) (1 (= normal)))
           do --))
 (TRANSLATE RULES)
```

At this point I realize that frequently I use this sort of advice, i.e., DO something and go on, so I add a new translation rule to handle it. With this rule, (solution2 : save y on solution) will become (TELL SOLUTION2 BEFORE DO SAVE Y ON SOLUTION).

```
(solution2 : save y on solution)
 SOLUTION2
```

Telling SOLUTION2 to SAVE Y.

```
(after solution1 : if value is null, then pop solution)
 SOLUTION1
```

After I come out of SOLUTION1, if its value is NIL, I want to remove X from SOLUTION.

```
(use solution1 after for solution2 after)
 (SOLUTION1 AFTER)
```

Similarly for SOLUTION2.

```
(after solve : mapc solution (print x))
 SOLVE
```

After I get done, I want to see SOLUTION.

```
solve ((at pencil county))
 (NIL AT PENCIL COUNTY)
 (NIL AT DESK COUNTY)
 (IN HOME Y)
 (((IN HOME Y) (AT Y COUNTY)) AT HOME COUNTY)
 (NIL AT HOME COUNTY)
 (NIL IN HOME COUNTY)
 (IN HOME COUNTY)
 (((IN HOME COUNTY)) AT HOME COUNTY)
 (AT HOME COUNTY)
 (((AT HOME COUNTY)) AT DESK COUNTY)
 (NIL IN DESK HOME)
 (IN DESK Y)
 (((IN DESK Y) (AT Y COUNTY)) AT DESK COUNTY)
 (AT DESK COUNTY)
 (((AT DESK COUNTY)) AT PENCIL COUNTY)
 (NIL IN PENCIL DESK)
 (IN PENCIL Y)
 (((IN PENCIL Y) (AT Y COUNTY)) AT PENCIL COUNTY)
 (AT PENCIL COUNTY)
 ((AT PENCIL COUNTY))
```

Now I ask the question at(pencil,county) again.

The first thing I notice is that I forgot to reverse SOLUTION and the deduction is reversed. I also notice that after answering the question (IN HOME COUNTY) with the fact (NIL IN HOME COUNTY), the program went on trying to get other answers via the statement (((IN HOME Y) (AT Y COUNTY)) AT HOME

COUNTY). I plan to remedy this. In order to evaluate the program's performance before and after this change, and others, I will make the program count the number of times it enters SOLUTION1 and SOLUTION2.

```
(change solve after
(replace solution with (reverse solution)))
 (SOLVE AFTER)
```

Reversing solution.

```
(solve : bind number to 0)
 SOLVE
```

Setting up a dummy variable NUMBER and binding it to 0.

```
(translate (- increment $1 -) as (- (setq 3 (add1 3)) -))
 (TRANSLATE RULES)
```

Defining what INCREMENT means.

```
(solution1 : increment number)
 SOLUTION1
```

Telling SOLUTION1 to INCREMENT NUMBER.

```
(solution2 : increment number)
 SOLUTION2
```

Similarly for SOLUTION2.

```
(after solve : (print cons number '(function calls)))
 SOLVE
```

Telling SOLVE to print NUMBER.

```
solve ((at pencil county))
 (AT PENCIL COUNTY)
 (((IN PENCIL Y) (AT Y COUNTY)) AT PENCIL COUNTY)
 (IN PENCIL Y)
 (NIL IN PENCIL DESK)
 (((AT DESK COUNTY)) AT PENCIL COUNTY)
 (AT DESK COUNTY)
 (((IN DESK Y) (AT Y COUNTY)) AT DESK COUNTY)
 (IN DESK Y)
 (NIL IN DESK HOME)
 (((AT HOME COUNTY)) AT DESK COUNTY)
 (AT HOME COUNTY)
 (((IN HOME COUNTY)) AT HOME COUNTY)
 (IN HOME COUNTY)
 (NIL IN HOME COUNTY)
 (NIL AT HOME COUNTY)
 (((IN HOME Y) (AT Y COUNTY)) AT HOME COUNTY)
 (IN HOME Y)
 (NIL AT DESK COUNTY)
 (NIL AT PENCIL COUNTY)

 (30 FUNCTION CALLS)

 ((AT PENCIL COUNTY))
```

I am now ready to tell the program not to look for additional answers to questions which do not contain any variables.

```
(solution1 : bind val to nil)
 SOLUTION1
```

I create the variable VAL and bind it to NIL. This will bind VAL to NIL each time SOLUTION1 is entered. Thus there will be a value for VAL associated with each question.

```
(after solution2 : (setq val value))
 SOLUTION2
```

When I leave SOLUTION2, I will set VAL to the value of SOLUTION2. The particular VAL that will be set will be the one associated with the question which created the statement that SOLUTION2 is considering currently.

```
(tell solution2, (before number advice),
if val and (variables x) is null, then quit)
 (SOLUTION2 BEFORE)
```

Now I tell SOLUTION2, before the advice which increments number* if VAL is true, which means I already obtained one answer to this question, and if (VARIABLES X) is NIL, which means that there are no variables in the question,** then it, SOLUTION2, should quit, i.e., return with NIL.

```
solve ((at pencil county))
 (AT PENCIL COUNTY)
 (((IN PENCIL Y) (AT Y COUNTY)) AT PENCIL COUNTY)
 (IN PENCIL Y)
 (NIL IN PENCIL DESK)
 (((AT DESK COUNTY)) AT PENCIL COUNTY)
 (AT DESK COUNTY)
 (((IN DESK Y) (AT Y COUNTY)) AT DESK COUNTY)
 (IN DESK Y)
 (NIL IN DESK HOME)
 (((AT HOME COUNTY)) AT DESK COUNTY)
 (AT HOME COUNTY)
 (((IN HOME COUNTY)) AT HOME COUNTY)
 (IN HOME COUNTY)
 (NIL IN HOME COUNTY)
 (NIL AT HOME COUNTY)
 (NIL AT DESK COUNTY)
 (NIL AT PENCIL COUNTY)

 (21 FUNCTION CALLS)

 ((AT PENCIL COUNTY))
```

Now I run the program again. Note that this time there are only 21 function calls, and that the extra effort at the end has been eliminated.

---

* I will not count as a bona fide call to SOLUTION2 if it is not actually entered.

** VARIABLES is a function in Black's original system.

English Output

From the thesis, I know that a list beginning with a list, including the empty list, is a statement, and lists headed by atoms represent questions. Why not make SOLVE talk English to me?

```
(define phrase (x) as flipq x
($1 (either ($1 / (variable)) ($1)) (either ($1 / (var-
iable)) ($1))) ((either 2 (any 1) (the 1)) is 1 (either
 3 (any 1) (the 1))) )
 (PHRASE)
```

I define the function PHRASE. PHRASE given (AT PENCIL COUNTY) will produce (THE PENCIL IS AT THE COUNTY). PHRASE given (AT PENCIL Y) will produce (THE PENCIL IS AT ANY Y).

```
(define question (x) as flipq phrase x ($ is $)
((= cr) 2 1 3 q)  )
 (QUESTION)
```

QUESTION is similar to PHRASE. QUESTION ((AT PENCIL COUNTY)) is (IS THE PENCIL AT THE COUNTY Q). QUESTION ((AT PENCIL Y)) is (IS THE PENCIL AT ANY Y Q).

```
(define clause (x) as flipq x ((either
(nil -)
(($1 (either ($1) -)) -)
(-) ))
((either
((** (= phrase 2)))
(if (** (= phrase (/c 1 1))) (either (/c 1 2) (and
   (** (= phrase 1))) -) (= comma) (** (= phrase 2))  )
((** (= question 1))) )   (= period) (= cr))  )
 (CLAUSE)
```

CLAUSE will transform questions or statements from Black's internal representation into English.

```
phrase ((at pencil county))
 (THE PENCIL IS AT THE COUNTY)
question ((at pencil y))
 (
 IS THE PENCIL AT ANY Y Q)
clause ((at pencil county))
 (
 IS THE PENCIL AT THE COUNTY Q .
)
clause ((nil at pencil county))
 (THE PENCIL IS AT THE COUNTY .
)
clause ((  ((at pencil y) (at y county)) at pencil county))
 (IF AND , THE PENCIL IS AT THE COUNTY .
)
```

A Bug in CLAUSE! Let's see if PHRASE is doing the right thing.

```
break (phrase t x)
 PHRASE
```

I BREAK on PHRASE, the "t" means that it will always break, and "x" that it will print the value of X.

```
clause ((  ((at pencil y) (at y county)) at pencil county))
 (BREAK IN PHRASE)
 ((AT PENCIL Y))
quit

PROCEED:
```

The trouble is an extra set of parentheses in certain situations.

I UNBREAK PHRASE and advise it that when (CDR X) is null, to take (CAR X) instead of X.

```
unbreak (phrase)
 PHRASE
(phrase : if (cdr x) is null, then (setq x car x))
 PHRASE
```

```
clause ((  ((at pencil y) (at y county)) at pencil county))
 (IF THE PENCIL IS AT ANY Y AND ANY Y IS AT THE COUNTY, THE
 PENCIL IS AT THE COUNTY .
```

Now CLAUSE works correctly.

```
(tell solve after, (instead of reverse advice),
mapc (append corpus reverse solution) (fancyprint clause x))
 (SOLVE AFTER)
```

Now instead of merely printing out the solution, I'll go through its solution and FANCYPRINT* the result of applying CLAUSE to each element on SOLUTION. I will also FANCYPRINT the corpus.

```
solve ((at pencil county))
 THE PENCIL IS IN THE DESK.
 THE DESK IS IN THE HOME.
 THE HOME IS IN THE COUNTY.
 IF ANY X IS IN ANY Y, ANY X IS AT ANY Y.
 IF ANY X IS IN ANY Y AND ANY Y IS AT ANY Z, ANY X IS AT
 ANY Z.

 IS THE PENCIL AT THE COUNTY Q.
 IF THE PENCIL IS IN ANY Y AND ANY Y IS AT THE COUNTY,
 THE PENCIL IS AT THE COUNTY.

 IS THE PENCIL IN ANY Y Q.
 THE PENCIL IS IN THE DESK.
 IF THE DESK IS AT THE COUNTY, THE PENCIL IS AT THE
 COUNTY.

 IS THE DESK AT THE COUNTY Q.
 IF THE DESK IS IN ANY Y AND ANY Y IS AT THE COUNTY, THE
 DESK IS AT THE COUNTY.

 IS THE DESK IN ANY Y Q.
 THE DESK IS IN THE HOME.
 IF THE HOME IS AT THE COUNTY, THE DESK IS AT THE COUNTY.

 IS THE HOME AT THE COUNTY Q.
 IF THE HOME IS IN THE COUNTY, THE HOME IS AT THE COUNTY.
```

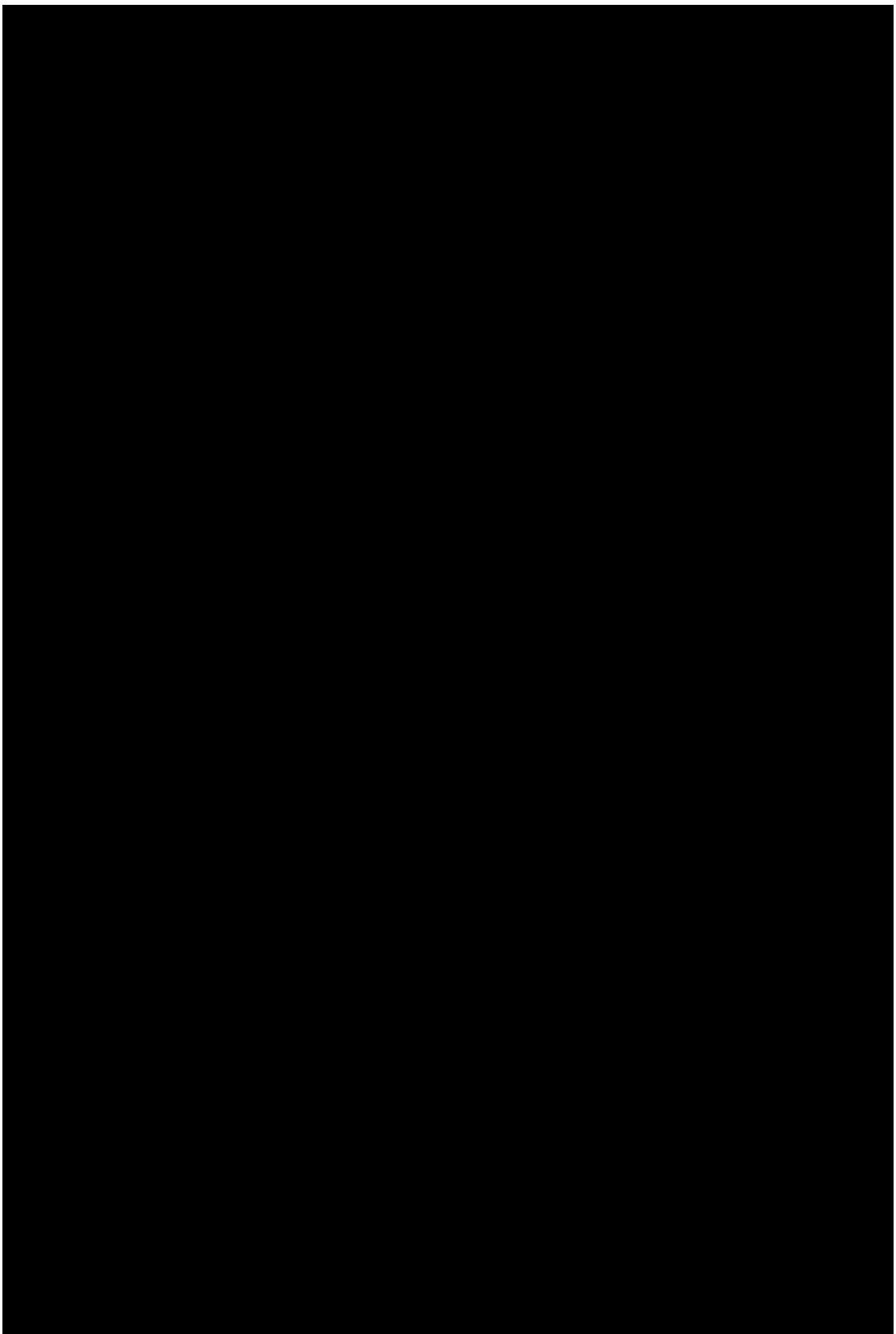
---

* FANCYPRINT is a trivial function which prints a list, suppressing initial and final parentheses without spacing before periods, commas, and colons.

```
 IS THE HOME IN THE COUNTY Q.
 THE HOME IS IN THE COUNTY.
 THE HOME IS AT THE COUNTY.
 THE DESK IS AT THE COUNTY.
 THE PENCIL IS AT THE COUNTY.

(21 FUNCTION CALLS)

((AT PENCIL COUNTY))
```



the advice will be appended at the end of the list of advice.

I need to be able to put something on the front of the list of advice.

```
(translate (tell $1 (either (first) ($1 / (atom) first))
$1) as    ($ system3 2 (= translate -1) (either ((= normal))
(1))))
 (TRANSLATE RULES)
```

Whenever I use the word FIRST it will mean to call SYSTEM3 instead of SYSTEM1.

```
(define system3 (what advice where) as
if (get what ' advised) is null, then (system1 what
   advice where),
else (prog2  put cons if advice is atomic then advice,
   else (cons ' advice advice) end get what where
   what where, what))
 (SYSTEM3)
```

SYSTEM3 will put the advice on the front.

```
(solution1 first : save x on record)
 SOLUTION1
```

Now I tell SOLUTION1 FIRST to save x on RECORD.

```
(solution2 first : save y on record)
 SOLUTION2
```

Similarly SOLUTION2,

```
(after solve : mapc (reverse record) (fancyprint clause x))
 SOLVE
```

and after SOLVE, to print RECORD.

Now I repeat the question. I can see that the program is on the right track. The last question it considered was "is the desk at any y." If it deduces "the desk is at the home," and the home is at the county," it will have deduced "the pencil is at the county."

```
solve ((at pencil county))
 THE PENCIL IS AT THE DESK.
 THE DESK IS AT THE HOME.
 THE HOME IS AT THE COUNTY.
 IF ANY X IS AT ANY Y AND ANY Y IS AT ANY Z, ANY X IS AT
 ANY Z.

 IS THE PENCIL AT THE COUNTY Q.
 IF THE PENCIL IS AT ANY Y AND ANY Y IS AT THE COUNTY,
 THE PENCIL IS AT THE COUNTY.

(8 FUNCTION CALLS)

 IS THE PENCIL AT THE COUNTY Q.
 IF THE PENCIL IS AT ANY Y AND ANY Y IS AT THE COUNTY,
 THE PENCIL IS AT THE COUNTY.

 IS THE PENCIL AT ANY Y Q.
 THE PENCIL IS AT THE DESK.
 IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
 PENCIL IS AT ANY Z.

 IS THE PENCIL AT ANY U Q.
 IF THE DESK IS AT THE COUNTY, THE PENCIL IS AT THE
 COUNTY.

 IS THE DESK AT THE COUNTY Q.
 IF THE DESK IS AT ANY Y AND ANY Y IS AT THE COUNTY, THE
 DESK IS AT THE COUNTY.

 IS THE DESK AT ANY Y Q.
NIL
```

Since I am going to have to manipulate the COUNTF parameter now set at 2, I would like to give it to SOLVE as one of its inputs.

```
(tell solve to bind n to (car x) and pop x)
 SOLVE
```

I tell SOLVE to BIND N to the first element of x, which will be this number, and to reset x to the rest of x.

```
(change solution1, (replace greaterp n1 $1 with n))
 (SOLUTION1 BEFORE)
```

I replace the "2" in GREATERP (countf something) 2, by N. I could also have said (REPLACE (COUNTF HISTORY ((SOLUTION1 $))) IS GREATER THAN 2 WITH (COUNTF HISTORY ((SOLUTION1 $))) IS GREATER THAN N).

Now try it with N set to 3.

```
solve ((3, at pencil county))
 THE PENCIL IS AT THE DESK.
 THE DESK IS AT THE HOME.
 THE HOME IS AT THE COUNTY.
 IF ANY X IS AT ANY Y AND ANY Y IS AT ANY Z, ANY X IS AT
 ANY Z.

 IS THE PENCIL AT THE COUNTY Q.
 IF THE PENCIL IS AT ANY Y AND ANY Y IS AT THE COUNTY,
 THE PENCIL IS AT THE COUNTY.

 IS THE PENCIL AT ANY Y Q.
 THE PENCIL IS AT THE DESK.
 IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
 PENCIL IS AT ANY Z.

 IS THE PENCIL AT ANY U Q.
 THE PENCIL IS AT THE DESK.
 IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

 IS THE DESK AT ANY Z Q.
 THE DESK IS AT THE HOME.
 THE PENCIL IS AT THE HOME.
 IF THE HOME IS AT THE COUNTY, THE PENCIL IS AT THE
 COUNTY.

 IS THE HOME AT THE COUNTY Q.
 THE HOME IS AT THE COUNTY.
 THE PENCIL IS AT THE COUNTY.

(17 FUNCTION CALLS)
```

```
IS THE PENCIL AT THE COUNTY Q.
IF THE PENCIL IS AT ANY Y AND ANY Y IS AT THE COUNTY,
THE PENCIL IS AT THE COUNTY.

IS THE PENCIL AT ANY Y Q.
THE PENCIL IS AT THE DESK.
IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
PENCIL IS AT ANY Z.

IS THE PENCIL AT ANY U Q.
THE PENCIL IS AT THE DESK.
IF THE PENCIL IS AT ANY Y AND ANY Y IS AT ANY Z, THE
PENCIL IS AT ANY Z.

IS THE PENCIL AT ANY Y Q.
IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

IS THE DESK AT ANY Z Q.
THE DESK IS AT THE HOME.
IF THE DESK IS AT ANY Y AND ANY Y IS AT ANY U, THE DESK
IS AT ANY U.

IS THE DESK AT ANY Y Q.
THE PENCIL IS AT THE HOME.
IF THE HOME IS AT THE COUNTY, THE PENCIL IS AT THE
COUNTY.

IS THE HOME AT THE COUNTY Q.
THE HOME IS AT THE COUNTY.
IF THE HOME IS AT ANY Y AND ANY Y IS AT THE COUNTY, THE
HOME IS AT THE COUNTY.
THE PENCIL IS AT THE COUNTY.
IF THE DESK IS AT THE COUNTY, THE PENCIL IS AT THE
COUNTY.
((AT PENCIL COUNTY))
```

The deduction took 17 function calls, and it considered (IS THE PENCIL AT ANY Y) 3 times.

I also try the (AT PENCIL Y) question (previously I was working with (AT PENCIL COUNTY)) to see how far I must allow it to run in order to produce all three answers.

```
solve ((1, at pencil y))
 THE PENCIL IS AT THE DESK.
 THE DESK IS AT THE HOME.
 THE HOME IS AT THE COUNTY.
 IF ANY X IS AT ANY Y AND ANY Y IS AT ANY Z, ANY X IS AT
 ANY Z.

 IS THE PENCIL AT ANY Y Q.
 THE PENCIL IS AT THE DESK.

(3 FUNCTION CALLS)

 IS THE PENCIL AT ANY Y Q.
 THE PENCIL IS AT THE DESK.
 IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
 PENCIL IS AT ANY Z.

 IS THE PENCIL AT ANY U Q.
((AT PENCIL DESK))
```

With N set to 1, it got one answer - the desk.

```
solve ((2, at pencil y))
 THE PENCIL IS AT THE DESK.
 THE DESK IS AT THE HOME.
 THE HOME IS AT THE COUNTY.
 IF ANY X IS AT ANY Y AND ANY Y IS AT ANY Z, ANY X IS AT
 ANY Z.

 IS THE PENCIL AT ANY Y Q.
 THE PENCIL IS AT THE DESK.
 IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
 PENCIL IS AT ANY Z.

 IS THE PENCIL AT ANY U Q.
 THE PENCIL IS AT THE DESK.
 IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

 IS THE DESK AT ANY Z Q.
 THE DESK IS AT THE HOME.
 THE PENCIL IS AT THE HOME.

(11 FUNCTION CALLS)

 IS THE PENCIL AT ANY Y Q.
 THE PENCIL IS AT THE DESK.
 IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
 PENCIL IS AT ANY Z.

 IS THE PENCIL AT ANY U Q.
 THE PENCIL IS AT THE DESK.
 IF THE PENCIL IS AT ANY Y AND ANY Y IS AT ANY Z, THE
 PENCIL IS AT ANY Z.

 IS THE PENCIL AT ANY Y Q.
 IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.
```

```
  IS THE DESK AT ANY Z Q.
  THE DESK IS AT THE HOME.
  IF THE DESK IS AT ANY Y AND ANY Y IS AT ANY U, THE DESK
  IS AT ANY U.

  IS THE DESK AT ANY Y Q.
  THE PENCIL IS AT THE HOME.
((AT PENCIL HOME) (AT PENCIL DESK))
```

With N at 2, it also got the home.

```
solve ((3, at pencil y))
  THE PENCIL IS AT THE DESK.
  THE DESK IS AT THE HOME.
  THE HOME IS AT THE COUNTY.
  IF ANY X IS AT ANY Y AND ANY Y IS AT ANY Z, ANY X IS AT
  ANY Z.

  IS THE PENCIL AT ANY Y Q.
  THE PENCIL IS AT THE DESK.
  IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
  PENCIL IS AT ANY Z.

  IS THE PENCIL AT ANY U Q.
  THE PENCIL IS AT THE DESK.
  IF THE PENCIL IS AT ANY Y AND ANY Y IS AT ANY Z, THE
  PENCIL IS AT ANY Z.

  IS THE PENCIL AT ANY Y Q.
  THE PENCIL IS AT THE DESK.
  IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

  IS THE DESK AT ANY Z Q.
  THE DESK IS AT THE HOME.
  THE PENCIL IS AT THE HOME.
  IF THE HOME IS AT ANY Z, THE PENCIL IS AT ANY Z.

  IS THE HOME AT ANY Z Q.
  THE HOME IS AT THE COUNTY.
  IF THE HOME IS AT ANY Y AND ANY Y IS AT ANY U, THE HOME
  IS AT ANY U.

  IS THE HOME AT ANY Y Q.
  THE PENCIL IS AT THE COUNTY.
  IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

  IS THE DESK AT ANY Z Q.
  THE DESK IS AT THE HOME.
  IF THE DESK IS AT ANY Y AND ANY Y IS AT ANY U, THE DESK
  IS AT ANY U.

  IS THE DESK AT ANY Y Q.
  THE DESK IS AT THE HOME.
  IF THE HOME IS AT ANY U, THE DESK IS AT ANY U.
```

```
IS THE HOME AT ANY U Q.
THE HOME IS AT THE COUNTY.
THE DESK IS AT THE COUNTY.
THE PENCIL IS AT THE COUNTY.
THE PENCIL IS AT THE HOME.

(39 FUNCTION CALLS)

IS THE PENCIL AT ANY Y Q.
THE PENCIL IS AT THE DESK.
IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
PENCIL IS AT ANY Z.

IS THE PENCIL AT ANY U Q.
THE PENCIL IS AT THE DESK.
IF THE PENCIL IS AT ANY Y AND ANY Y IS AT ANY Z, THE
PENCIL IS AT ANY Z.

IS THE PENCIL AT ANY Y Q.
THE PENCIL IS AT THE DESK.
IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
PENCIL IS AT ANY Z.

IS THE PENCIL AT ANY U Q.
IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

IS THE DESK AT ANY Z Q.
THE DESK IS AT THE HOME.
IF THE DESK IS AT ANY Y AND ANY Y IS AT ANY U, THE DESK
IS AT ANY U.

IS THE DESK AT ANY Y Q.
THE PENCIL IS AT THE HOME.
IF THE HOME IS AT ANY Z, THE PENCIL IS AT ANY Z.

IS THE HOME AT ANY Z Q.
THE HOME IS AT THE COUNTY.
IF THE HOME IS AT ANY Y AND ANY Y IS AT ANY U, THE HOME
IS AT ANY U.

IS THE HOME AT ANY Y Q.
THE HOME IS AT THE COUNTY.
IF THE HOME IS AT ANY U AND ANY U IS AT ANY Z, THE HOME
IS AT ANY Z.

IS THE HOME AT ANY U Q.
IF THE COUNTY IS AT ANY U, THE HOME IS AT ANY U.

IS THE COUNTY AT ANY U Q.
IF THE COUNTY IS AT ANY Y AND ANY Y IS AT ANY Z, THE
COUNTY IS AT ANY Z.

IS THE COUNTY AT ANY Y Q.
THE PENCIL IS AT THE COUNTY.
IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

IS THE DESK AT ANY Z Q.
THE DESK IS AT THE HOME.
IF THE DESK IS AT ANY Y AND ANY Y IS AT ANY U, THE DESK
IS AT ANY U.
```

```
IS THE DESK AT ANY Y Q.
THE DESK IS AT THE HOME.
IF THE DESK IS AT ANY U AND ANY U IS AT ANY Z, THE DESK
IS AT ANY Z.

IS THE DESK AT ANY U Q.
IF THE HOME IS AT ANY U, THE DESK IS AT ANY U.

IS THE HOME AT ANY U Q.
THE HOME IS AT THE COUNTY.
IF THE HOME IS AT ANY Y AND ANY Y IS AT ANY Z, THE HOME
IS AT ANY Z.

IS THE HOME AT ANY Y Q.
THE DESK IS AT THE COUNTY.
THE PENCIL IS AT THE COUNTY.
THE PENCIL IS AT THE HOME.
((AT PENCIL HOME) (AT PENCIL COUNTY) (AT PENCIL COUNTY) (AT
PENCIL DESK))
```

With N at 3, it got the county, as it should, but it took 39 function calls, because it kept reconsidering the same questions until it ran out of room. Only then did it abandon this question and proceed to the next one.

What I really want to do is note when a question repeats and take the answers found so far. I can do this because HISTORY is available and I can look back on it and find VAL, which has all of the answers bound to it.

Instead of the countf advice, I will use a FLIP rule which will look for SOLUTION1 on the HISTORY list, provided its argument x, matches the current x. In this case, it will return with the value of VAL.

```
(tell solution1, (instead of countf advice), (flip1 history
'(- (solution1 -) (val -) (x $ / (matches (= x))) -)
'(((/t 3 2))) history))
 (SOLUTION1 BEFORE)


(define matches (x y) as
if x is null, then y is null,
if (car x) is equal to (car y) or
(variable car x) and (variable car y),
  then (matches cdr x cdr y))
 (MATCHES)
```

Two questions will match if they are identical except for substitution of variables. Black's function VARIABLE is true if its input is the name of a variable, e.g., X, Y, U, V, etc.

```
(change solve, (delete n (backto advice) up1))
 (SOLVE BEFORE)
```

I don't need the advice concerning N.

```
solve ((at pencil y))
  THE PENCIL IS AT THE DESK.
  THE DESK IS AT THE HOME.
  THE HOME IS AT THE COUNTY.
  IF ANY X IS AT ANY Y AND ANY Y IS AT ANY Z, ANY X IS AT
  ANY Z.

  IS THE PENCIL AT ANY Y Q.
  THE PENCIL IS AT THE DESK.
  IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
  PENCIL IS AT ANY Z.

  IS THE PENCIL AT ANY U Q.
  IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

  IS THE DESK AT ANY Z Q.
  THE DESK IS AT THE HOME.
  IF THE DESK IS AT ANY Y AND ANY Y IS AT ANY U, THE DESK
  IS AT ANY U.

  IS THE DESK AT ANY Y Q.
  IF THE HOME IS AT ANY U, THE DESK IS AT ANY U.

  IS THE HOME AT ANY U Q.
  THE HOME IS AT THE COUNTY.
  IF THE HOME IS AT ANY Y AND ANY Y IS AT ANY Z, THE HOME
  IS AT ANY Z.
  THE DESK IS AT THE COUNTY.
  THE PENCIL IS AT THE COUNTY.
  THE PENCIL IS AT THE HOME.

 (17 FUNCTION CALLS)

  IS THE PENCIL AT ANY Y Q.
  THE PENCIL IS AT THE DESK.
  IF THE PENCIL IS AT ANY U AND ANY U IS AT ANY Z, THE
  PENCIL IS AT ANY Z.

  IS THE PENCIL AT ANY U Q.
  IF THE DESK IS AT ANY Z, THE PENCIL IS AT ANY Z.

  IS THE DESK AT ANY Z Q.
  THE DESK IS AT THE HOME.
  IF THE DESK IS AT ANY Y AND ANY Y IS AT ANY U, THE DESK
  IS AT ANY U.
```

```
IS THE DESK AT ANY Y Q.
IF THE HOME IS AT ANY U, THE DESK IS AT ANY U.

IS THE HOME AT ANY U Q.
THE HOME IS AT THE COUNTY.
IF THE HOME IS AT ANY Y AND ANY Y IS AT ANY Z, THE HOME
IS AT ANY Z.

IS THE HOME AT ANY Y Q.
IF THE COUNTY IS AT ANY Z, THE HOME IS AT ANY Z.

IS THE COUNTY AT ANY Z Q.
IF THE COUNTY IS AT ANY Y AND ANY Y IS AT ANY U, THE
COUNTY IS AT ANY U.

IS THE COUNTY AT ANY Y Q.
THE DESK IS AT THE COUNTY.
THE PENCIL IS AT THE COUNTY.
THE PENCIL IS AT THE HOME.
((AT PENCIL HOME) (AT PENCIL COUNTY) (AT PENCIL DESK))
```

Now the deduction requires only 17 function calls, and looks reasonable!

CHAPTER 6

EXPERIMENTS WITH A PROBLEM SOLVER

The central aim of the General Problem Solver of Newell, Simon and Shaw[38] was to divorce problem solving techniques and heuristics from any task environment, and thus construct a program that was truly general. A system was constructed that succeeded in proving theorems in logic, and solving problems such as the cannibal and missionary problem. However, the system grew so massive and cumbersome, and the effort involved in making modifications so enormous, that it has become more or less frozen. (Newell has informed me that after some time away from the program, it takes him weeks just to "get into the listing" and remember what the program does.)

I thought it might be worthwhile to use PILOT to construct a system with the same goals as GPS, i.e., flexibility and generality, although not as complex. I started with a minimal configuration and used PILOT to make modifications as I went along. In this way I did not give much forethought to the design of the system, but allowed it to develop as the experimentation proceeded. The next section summarizes what happened, and the following section contains a protocol which is an extract from my sessions at the console.

Summary of Experiments

The basic design of the system is illustrated in the accompanying flow chart. I implemented this flow chart by dividing the various tasks among five functions, thus facilitating making subsequent changes with advice. These functions are MOVES, GOALP, GPS, MAKE, AND PROGRESS. MOVES generates a list of moves for any given situation. GOALP recognizes when the problem has been solved. The main loop of the program is GPS--MAKE--PROGRESS--GPS. GPS* is the executive routine which calls MOVES, selects the first move on the list of possible moves, and calls MAKE. MAKE makes the move, i.e., performs the necessary changes in the problem representation, and calls PROGRESS. PROGRESS checks whether the problem has been solved by calling GOALP, and, if not, calls GPS with the new position.

The first problem I attempted to solve with the system was the cannibal and missionary problem. In this problem, three cannibals and three missionaries are on one side of a river with a boat that can carry only two men. The object is to transport everyone across the river. The catch is that if there are more cannibals than missionaries on any side at any time, the cannibals will eat the missionaries. This is undesirable. It is also assumed that the boat will not float across the river by itself, i.e., someone has to be in it to take it across.

I set up the problem using four variables: SIDE1, SIDE2, FROM, and TO. SIDE1 would represent the contingent on the near side of the river, and SIDE2 those on the far side. FROM and

---

* I hope Messrs. Newell, Simon, and Shaw will forgive me for naming my program after theirs.

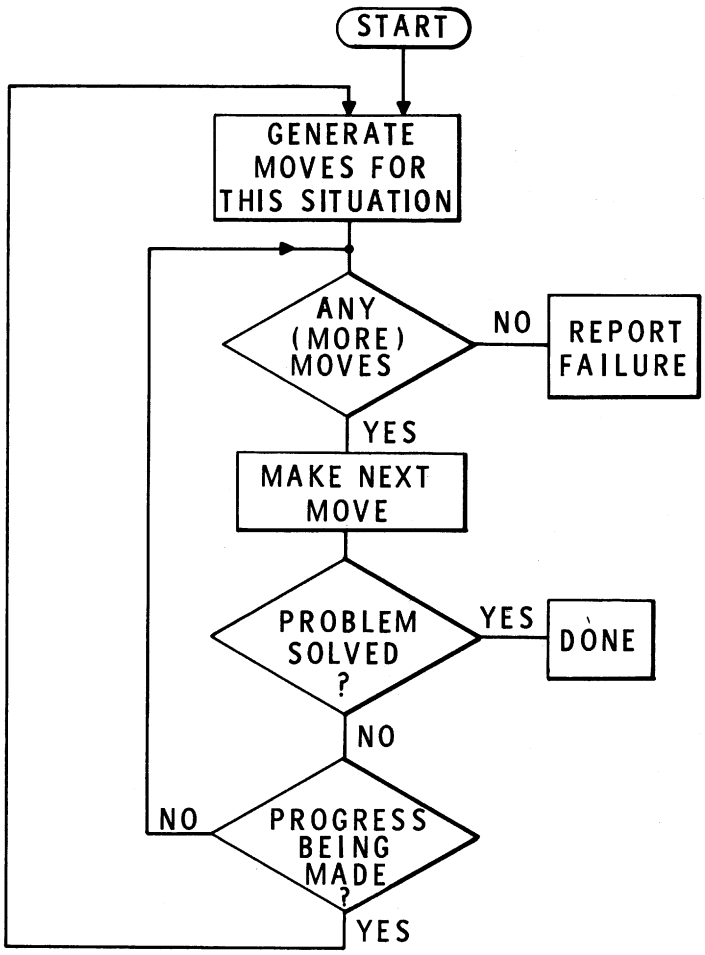


FIG. 4 A SIMPLE PROBLEM SOLVER

TO would represent the direction of transfer, in other words, the location of the boat. I advised GOALP of the terminal conditions, and told MOVES to return with MOVE1 and MOVE2, corresponding to moving 1 person and 2 people. (I had to define the operation of moving appropriately.) I then advised MAKE to make the appropriate changes in FROM, TO, SIDE1, and SIDE2, and instructed PROGRESS to quit if the cannibals outnumbered the missionaries. The only thing remaining was to ensure that GPS did not loop, i.e., send a cannibal across, bring him back, send him across, bring him back, etc. I advised GPS to avoid looping by saving the positions encountered, searching this list of positions, and terminating a branch when a position repeated. With this set of advice, GPS solved the problem.

Unfortunately, as one can see from the interaction shown below, solving the problem simply meant that ten seconds after input,GPS printed *T*, indicating the problem had been solved. This was not very informative. Therefore, I modified the program to count the number of moves it considered, and to print the solution. At this stage, I decided to see if I could get a nice English output.

I defined a function PLURAL, which took the plural of nouns, and by advice, enabled it to handle the plurals of words like both CANNIBAL and MISSIONARY - drop the "y" and add "i e s," etc. I defined a function PHRASE, which took a list of the form (C M C M M), C standing for CANNIBAL and M for MISSIONARY, and produced (TWO CANNIBALS AND THREE MISSIONARIES). (This was necessary because there was no guarantee that the representation would be sorted, and indeed it usually wasn't.) When I got PHRASE working, I had the program print the solution and then

added a facility to have it print out each move considered. Since nothing was built into the program to distinguish one cannibal from another (MOVE1 simply meant take 1 person and move him), the program would attempt to send across one cannibal, then to bring him back - that then being the only legal move - realize that it was back where it started, abandon this line of attack, and generate as its next move, sending across the next cannibal, etc. It was obvious that heuristics were needed.

As a first heuristic I told GPS that if it was trying to send people across, i.e., going FROM SIDE1 and TO SIDE2, then it should try to send as many men as possible, i.e., to consider moving 2 before moving 1. This was to avoid fruitless considerations of trying to send each one of the original six people across before trying combinations of two. This heuristic reduced the number of moves attempted from 68 to 35. I then added a second heuristic which had the effect of making the program realize that once it had tried sending across a particular boatload, and failed, it should not try the same move again. This reduced the number of moves considered to 20. The length of the solution in each case was 11 moves, which is the minimum number required.

Since GPS was supposed to be a general problem solving program, I now asked it to solve the fox, goose, and corn problem. In this problem, a farmer wants to carry a fox, a goose, and some corn to the barn, but can't leave the fox alone with the goose or the goose alone with the corn. In addition, he can only carry one object at a time.

Since this problem was similar to the cannibal and missionary problem, I was able to carry over much of the advice already given to GPS, GOALP, MOVES, MAKE, and PROGRESS, making only a few modifications. GPS then solved the problem.

Professor Minsky suggested that I try the cannibal and missionary problem again, this time with a boat that could carry three people. This modification turned out to be easy to achieve by advising MOVES. GPS only considered 12 moves to find the solution, now requiring only five moves.

I decided I would now like to be able to solve the problem using the number of missionaries and cannibals as input parameters. I modified PROGRESS, changing the advice that checked on the missionaries' safety to work with any size population. I then gave the program the problem with 4 missionaries and 4 cannibals, which can't be solved with a two man boat, as the program discovered. This problem can be solved with a larger boat,and the program found solution for the modified problem.

I decided I would like to specify the size of the boat as an input parameter also. After doing this, I asked the program to SOLVE (CANNIBAL AND MISSIONARY PROBLEM FOR 3 IN A BOAT AND FOR 4 CANNIBALS AND 5 MISSIONARIES).

At this point, the program ran out of space, primarily because I had, resident in core, all of FLIP, the SYSTEM functions, and the EDIT package, in addition to the problem solving program. I made room by removing the EDIT functions and continued to a solution of the problem. When I finished, I enabled the system

to make room in the future when it required it, by telling the system whenever there were less than 500 words of free storage left, to remove the least essential package. GPS then solved a number of other problems.

I made two more interesting modifications to the program. First, I advised it how to solve (HOW BIG A BOAT DO YOU NEED FOR 4 CANNIBALS AND 4 MISSIONARIES). This was a change conceptually simple, since it only involved GPS calling itself with different size boats until one was found that worked. However, I made a more sophisticated revision that involved a problem in which some of the missionary population might not be eaten by cannibals, even though outnumbered. I called such a missionary TARZAN, and asked the system to solve problems such as (HOW BIG A BOAT DO YOU NEED FOR 3 MISSIONARIES, 1 TARZAN, 4 CANNIBALS) - answer 2.

Protocol

```
(define gps as prog (x y)
setq x moves
g1    if x is null then (return nil) end
setq y valueof car x
g2    if y is null then (go g3),
      if (make car y) then (return t)  end
pop y
go g2
g3  pop x
go g1)
 (GPS)
```

This is the definition of GPS. GPS calls MOVES which returns with a list of the move types, not the moves themselves. GPS then computes all of the moves corresponding to a particular type, and runs through them calling MAKE on each one.

```
(define moves as)
 (MOVES)

(define make (move) as progress)
 (MAKE)

(define progress as goalp or gps)
 (PROGRESS)

(define goalp as)
 (GOALP)
```

Definition of MOVES, MAKE, PROGRESS, GOALP. MOVES and GOALP are defined as nothing - which means they return NIL. MAKE is a function of one variable - its name being MOVE.

```
(define solve (fexpr) as
if (get ' start csetq normal car 1) is null,
    then '(dont know how),
if (csetq history list cons ' solve 1) then (start nil))
 (SOLVE)
```

SOLVE takes the statement of the problem and determines whether the problem can be solved. It looks on the property list of START for advice on this problem. (Problems are labeled by the first word in the statement, for example CANNIBAL.) If there is none, SOLVE returns (DONT KNOW HOW). Otherwise it calls START to begin solving the problem. SOLVE also sets the NORMAL mode to the problem name so that further advice is interpreted in the context of this problem.

```
(define start (hist) as gps)
 (START)
```

START performs the initialization and calls GPS. START has one variable, HIST, which may be used for saving information to be printed out at the end.

Since I would like to use the same program for several different problems, I will prepare for different entrance points corresponding to the various problems. This is done by placing a computation which will produce the advice at the canonical entry point labeled BEFORE, instead of the actual list of advice. This is the role of the function SETUP.

```
(define setup (x) as mapc x (nconc x list ' before
'(lambda (y) (get (caadr y) normal))))
 (SETUP)
```

SETUP places under the property BEFORE, the S-expression (LAMBDA (X) (GET (CAADR Y) NORMAL)), which is evaluated by ADVISE. This will get the list of advice from the correct property. As described in Chapter 3, the input to this LAMBDA expression is the HISTORY list, and CAADR of the HISTORY list is always the name of the function just entered.

```
setup ((gps moves make progress goalp start))
 NIL


(translate (start with $1 $1 (repeat $1 $1)) as
(tell start to bind 3 to ($* quote 4)
(repeat n and bind 1 to ($* quote 2)) and nil))
 (TRANSLATE RULES)
```

This rule causes instructions of the form (START WITH uuu vvv xxx yyy ...) to be transformed into advice for START which will perform the appropriate operation of binding uuu to vvv, xxx to yyy, etc. This advice corresponds to the initialization process.

I now try to SOLVE (CANNIBAL AND MISSIONARIES) and GPS responds (DONT KNOW HOW) because under the property CANNIBAL on the property list of START there is no advice - yet. NORMAL is set to CANNIBAL.

```
solve (cannibal and missionaries)
 (DONT KNOW HOW)
```

```
(start with side1 (m m m c c c), side2 nil, to side2,
from side1)
 START
```

Start with side1 (m m m c c c), go to side2 from side1.

```
(tell goalp, return with side1 is null)
 GOALP
```

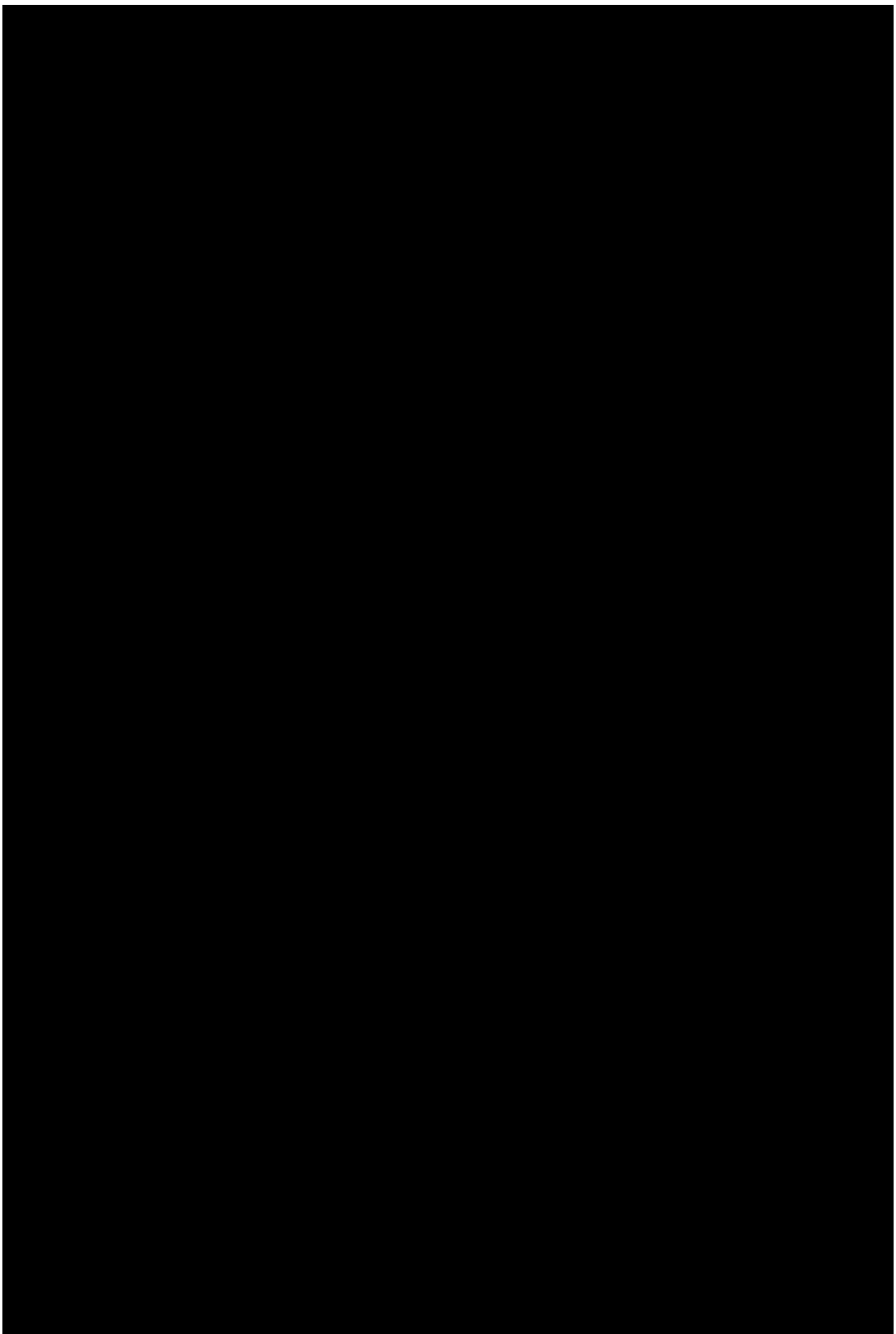
Final condition - no one left on side1.

```
(tell moves, return with '((move1) (move2)))
 MOVES
```

Moves.

```
(define move1 as alltran valueof from '($1) '((2) 1 3))
 MOVE1)
```

MOVE1 goes through VALUEOF FROM and makes a list containing a move corresponding to every single element on FROM. In other words, if FROM is SIDE1, and SIDE1 is (M C M C), then the value of MOVE1 is (((M) C M C), ((C) M M C), ((M) M C C), ((C) M C M)). Each member corresponds to a move, namely the one in which the first item is moved, leaving the rest. MOVE1 is easily defined using a FLIP function ALLTRAN. ALLTRAN yields all possible



a point in the solution in which the position is repeated with respect to the values of SIDE1 and SIDE2, but the boat is on a different side.

```
(after gps : pop hist)
 GPS
```

After leaving GPS, I have to pop hist.

```
(tell progress, if searchf hist (((= from) $ /
(setequal (= side2)))) then quit)
 PROGRESS
```

This advice tells progress to search through HIST looking for an element whose first member is equal to the value of FROM, and the rest of which is equal, in the set terminology sense, to the value of SIDE2. We must use set equality because the representation may have become rearranged.

```
solve (cannibal and missionaries)
 *T*
```

Now it can solve the problem.

Unfortunately, GPS gives me little information about what it did, so I set up two more variables, NUMBER, and SOLUTION.

```
(start with number 0, solution nil)
 START
```

```
(make : increment number)
 MAKE
```

At MAKE, I increment number.

```
(after goalp : if value then (setq solution history))
 GOALP
```

After GOALP, if its value is not NIL, which means the answer has been found, I save the HISTORY list on SOLUTION.

```
(after start : if value is not equal to '(dont know
how), then (printred cons number '(moves considered)))
 START
```

When I get back to START, if the value is not (DONT KNOW HOW), I print the number of moves considered,

```
(after start : mapc (listf reverse solution
'((move $)) '(((/t 2 2))) nil)  (print x))
 START
```

and a summary of what they were. Here I use another FLIP function LISTF, to look through the HISTORY list and make a list of all of the bindings of the variable MOVE (which is the name of the argument of MAKE).

```
solve (cannibal and missionaries)
 (68 MOVES CONSIDERED)
 ((M C) M M C C)
 ((M) C)
 ((C C) M M M)
 ((C) C C)
 ((M M) C M)
 ((M C) M C)
 ((M M) C C)
 ((C) M M M)
 ((C C) C)
 ((C) C M M M)
 ((C C))
 *T*
```

This is the solution. The first move was to take a missionary and a cannibal across, leaving two missionaries and two cannibals on the near side. Then a missionary came back leaving a lone cannibal on the far side. Next two cannibals went across, etc.

English Output

```
(define plural1 (x y) as prog3 clearbuff
mapc (append x y) (pack x), intern mknam)
 (PLURAL1)
```

PLURAL1 is a function which takes its two arguments and makes one word out of them.

```
(define plural (x) as plural1 explode x '(s))
 (PLURAL)
```

PLURAL calls PLURAL1 with its input and "(s)." Thus PLURAL (CANNIBAL) IS CANNIBALS.

```
plural (cannibal)
 CANNIBALS
plural (missionary)
 MISSIONARYS
```

```
(plural1 : if (last x) is equal to '(y),
then to (rlast x) and (setq y '(i e s)))
 PLURAL1
```

Telling PLURAL1 if the last letter is a "y," is should RLAST, remove the last letter, and use "I E S" instead of "S."

```
plural (missionary)
 MISSIONARIES
```

Now it works correctly.

```
(define phrase (x y) as if x is null, then '(nobody),
else (transform sublis y x '(
((-- $1 / (atom) (repeat $ (/t 2)) --)
     (-- 4 (repeat 1) ((= (car n)) 2)) top)
(((repeat ((either ((= 1) $1) ((= 2) $1) ((= 3) $1)
($1 $1)))))
     ((repeat (either (/c 1 1) (a 2) (two (= plural 2))
        (three (= plural 2)) (1 (= plural 2)))
        (= comma))))
(($2 $1) (1) exit)
(($2 $1 $2 $1 (1 and 3) exit)
(((back 3) $2 $1) (1 and 2))  ))  )
 (PHRASE)
```

PHRASE sorts the people on a side, substituting their proper names, i.e., cannibal for "c" and missionary for "m," and then makes a nice phrase out of it.

I test PHRASE.

```
phrase ((c) ((c . cannibal)))
 (A CANNIBAL)
phrase ((c m) ((c . cannibal) (m . missionary)))
 (A CANNIBAL AND A MISSIONARY)
phrase ((c m c m m) ((c . cannibal) (m . missionary)))
 (TWO (CANNIBAL)S AND THREE (MISSIONARY)S)
```

A Bug, because I have extra parentheses.

```
(plural : if x is not atomic, then (setq x car x))
 PLURAL
```

Fix the BUG

```
phrase ((c m c m m) ((c . cannibal) (m . missionary)))
 (TWO CANNIBALS AND THREE MISSIONARIES)
phrase ((c m c m m m 1) ((c . cannibal) (m . missionary)
(1 . lion)))
 (TWO CANNIBALS , 4 MISSIONARIES , AND A LION)
```

and it works correctly.

```
(tell start after, (instead of listf advice),
if value and (cadr solve) is equal to ' and, then
(fancyprint listf reverse solution '((move ($) $) $
   (either ((from . side1)) ((from . side2))))
'(bring (** (= phrase (/t 2 2 1) (= a)))
   (either (back) (across)) (= comma) leaving
   (** (= phrase (/t 2 3) (= a))) on
   (either (side2) (side1)) (= period) (= cr) (= cr))
'((a (c . cannibal) (m . missionary)))  ))
 (START AFTER)
```

I replace the LISTF advice with advice for producing fancy output. The result is shown below.

```
solve (cannibal and missionaries)

 (68 MOVES CONSIDERED)

  BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING TWO
  MISSIONARIES AND TWO CANNIBALS ON SIDE1.

  BRING A MISSIONARY BACK, LEAVING A CANNIBAL ON SIDE2.

  BRING TWO CANNIBALS ACROSS, LEAVING THREE MISSIONARIES
  ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON SIDE2.

  BRING TWO MISSIONARIES ACROSS, LEAVING A CANNIBAL AND
  A MISSIONARY ON SIDE1.

  BRING A MISSIONARY AND A CANNIBAL BACK, LEAVING A
  MISSIONARY AND A CANNIBAL ON SIDE2.

  BRING TWO MISSIONARIES ACROSS, LEAVING TWO CANNIBALS
  ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING THREE MISSIONARIES ON
  SIDE2.

  BRING TWO CANNIBALS ACROSS, LEAVING A CANNIBAL ON
  SIDE1.

  BRING A CANNIBAL BACK, LEAVING A CANNIBAL AND THREE
  MISSIONARIES ON SIDE2.

  BRING TWO CANNIBALS ACROSS, LEAVING NOBODY ON SIDE1.

 *T*
```

In order to evaluate new heuristics, I also need a facility for printing each move as it is considered. I will modify MAKE, so that if the word "PROBLEM" appears in the input, MAKE will print each move in a nice format.

```
(make : if ' problem is a member of solve, then
(fancyprint construct1 nil '(bring (** (= (phrase car
move a)))
   (= (sublis '((side1 . back) (side2 . across))
   from)) (= cr)
side1 : (** (= (phrase side1 a))) (= cr)
side2 : (** (= (phrase side2 a))) (= cr) (= cr))
    cons '(a (c . cannibal) (m . missionary)) history))
 MAKE


solve (cannibal problem)
  BRING A MISSIONARY ACROSS
  SIDE1:  TWO MISSIONARIES AND THREE CANNIBALS
  SIDE2:  A MISSIONARY

  BRING A MISSIONARY ACROSS
  SIDE1:  TWO MISSIONARIES AND THREE CANNIBALS
  SIDE2:  A MISSIONARY

  BRING A MISSIONARY ACROSS
  SIDE1:  TWO MISSIONARIES AND THREE CANNIBALS
  SIDE2:  A MISSIONARY

  BRING A CANNIBAL ACROSS
  SIDE1:  THREE MISSIONARIES AND TWO CANNIBALS
  SIDE2:  A CANNIBAL

  BRING A CANNIBAL BACK
  SIDE1:  THREE CANNIBALS AND THREE MISSIONARIES
  SIDE2:  NOBODY

  BRING A CANNIBAL ACROSS
  SIDE1:  THREE MISSIONARIES AND TWO CANNIBALS
  SIDE2:  A CANNIBAL

  BRING A CANNIBAL BACK
  SIDE1:  THREE CANNIBALS AND THREE MISSIONARIES
  SIDE2:  NOBODY

  BRING A CANNIBAL ACROSS
  SIDE1:  THREE MISSIONARIES AND TWO CANNIBALS
  SIDE2:  A CANNIBAL

  BRING A CANNIBAL BACK
  SIDE1:  THREE CANNIBALS AND THREE MISSIONARIES
  SIDE2:  NOBODY
```

```
BRING TWO MISSIONARIES ACROSS
SIDE1:  A MISSIONARY AND THREE CANNIBALS
SIDE2:  TWO MISSIONARIES

BRING TWO MISSIONARIES ACROSS
SIDE1:  A MISSIONARY AND THREE CANNIBALS
SIDE2:  TWO MISSIONARIES

BRING A MISSIONARY AND A CANNIBAL ACROSS
SIDE1:  TWO MISSIONARIES AND TWO CANNIBALS
SIDE2:  A MISSIONARY AND A CANNIBAL

            etc.
```

These are the first 12 moves considered. Note that the program does not assume that any one missionary is different from any other.

The first heuristic is to try MOVE2 before MOVE1, when the boat is going <u>across</u> the river, but keep MOVE1 first when coming back. I instruct MOVES to reverse its value if TO is equal to SIDE2.

```
(tell moves after, if to is equal to ' side2, then
return with (reverse value))
 MOVES


solve (cannibal)
 (35 MOVES CONSIDERED)
 *T*
```

Now the number of moves is reduced to 35. The next heuristic is to save the moves considered at each ply, and not attempt one which is SETEQUAL to a move considered before. SETEQUAL must be used because the move (M C) should eliminate (C M).

```
(gps : bind moves to nil)
 GPS
```

Setting up the dummy variable MOVES.

```
(tell make first, if searchp moves (setequal (car
move)), then quit, else do save (car move) on moves)
 MAKE
```

Telling MAKE, FIRST, to search MOVES, and if it finds something which is SETEQUAL to (CAR MOVE), then quit. Otherwise, save (CAR MOVE). This cuts the solution down to 20 moves, reproduced here in full.

```
solve (cannibal and missionary problem)
  BRING TWO MISSIONARIES ACROSS
  SIDE1:  A MISSIONARY AND THREE CANNIBALS
  SIDE2:  TWO MISSIONARIES

  BRING A MISSIONARY AND A CANNIBAL ACROSS
  SIDE1:  TWO MISSIONARIES AND TWO CANNIBALS
  SIDE2:  A MISSIONARY AND A CANNIBAL

  BRING A MISSIONARY BACK
  SIDE1:  THREE MISSIONARIES AND TWO CANNIBALS
  SIDE2:  A CANNIBAL

  BRING TWO MISSIONARIES ACROSS
  SIDE1:  A MISSIONARY AND TWO CANNIBALS
  SIDE2:  TWO MISSIONARIES AND A CANNIBAL

  BRING A MISSIONARY AND A CANNIBAL ACROSS
  SIDE1:  TWO MISSIONARIES AND A CANNIBAL
  SIDE2:  A MISSIONARY AND TWO CANNIBALS

  BRING TWO CANNIBALS ACROSS
  SIDE1:  THREE MISSIONARIES
  SIDE2:  THREE CANNIBALS

  BRING A CANNIBAL BACK
  SIDE1:  A CANNIBAL AND THREE MISSIONARIES
  SIDE2:  TWO CANNIBALS

  BRING A CANNIBAL AND A MISSIONARY ACROSS
  SIDE1:  TWO MISSIONARIES
  SIDE2:  THREE CANNIBALS AND A MISSIONARY

  BRING TWO MISSIONARIES ACROSS
  SIDE1:  A CANNIBAL AND A MISSIONARY
  SIDE2:  TWO MISSIONARIES AND TWO CANNIBALS

  BRING A MISSIONARY BACK
  SIDE1:  TWO MISSIONARIES AND A CANNIBAL
  SIDE2:  A MISSIONARY AND TWO CANNIBALS
```

```
BRING A CANNIBAL BACK
SIDE1:  TWO CANNIBALS AND A MISSIONARY
SIDE2:  TWO MISSIONARIES AND A CANNIBAL

BRING TWO MISSIONARIES BACK
SIDE1:  THREE MISSIONARIES AND A CANNIBAL
SIDE2:  TWO CANNIBALS

BRING A MISSIONARY AND A CANNIBAL BACK
SIDE1:  TWO MISSIONARIES AND TWO CANNIBALS
SIDE2:  A MISSIONARY AND A CANNIBAL

BRING A MISSIONARY AND A CANNIBAL ACROSS
SIDE1:  A CANNIBAL AND A MISSIONARY
SIDE2:  TWO MISSIONARIES AND TWO CANNIBALS

BRING TWO MISSIONARIES ACROSS
SIDE1:  TWO CANNIBALS
SIDE2:  THREE MISSIONARIES AND A CANNIBAL

BRING A MISSIONARY BACK
SIDE1:  A MISSIONARY AND TWO CANNIBALS
SIDE2:  TWO MISSIONARIES AND A CANNIBAL

BRING A CANNIBAL BACK
SIDE1:  THREE CANNIBALS
SIDE2:  THREE MISSIONARIES

BRING TWO CANNIBALS ACROSS
SIDE1:  A CANNIBAL
SIDE2:  TWO CANNIBALS AND THREE MISSIONARIES

BRING A CANNIBAL BACK
SIDE1:  TWO CANNIBALS
SIDE2:  A CANNIBAL AND THREE MISSIONARIES

BRING TWO CANNIBALS ACROSS
SIDE1:  NOBODY
SIDE2:  THREE CANNIBALS AND THREE MISSIONARIES

(20 MOVES CONSIDERED)

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
TWO MISSIONARIES AND TWO CANNIBALS ON SIDE1.

BRING A MISSIONARY BACK, LEAVING A CANNIBAL ON
SIDE2.

BRING TWO CANNIBALS ACROSS, LEAVING THREE MISSIONARIES
ON SIDE1.

BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON
SIDE2.

BRING TWO MISSIONARIES ACROSS, LEAVING A CANNIBAL
AND A MISSIONARY ON SIDE1.

BRING A MISSIONARY AND A CANNIBAL BACK, LEAVING
A MISSIONARY AND A CANNIBAL ON SIDE2.

BRING TWO MISSIONARIES ACROSS, LEAVING TWO CANNIBALS
ON SIDE1.
```

```
BRING A CANNIBAL BACK, LEAVING THREE MISSIONARIES ON
SIDE2.

BRING TWO CANNIBALS ACROSS, LEAVING A CANNIBAL ON SIDE1.

BRING A CANNIBAL BACK, LEAVING A CANNIBAL AND THREE
MISSIONARIES ON SIDE2.

BRING TWO CANNIBALS ACROSS, LEAVING NOBODY ON SIDE1.

*T*
```

Now I try the fox, gooose, and corn problem.

```
solve (fox, goose, and corn problem)
 (DONT KNOW HOW)
```

I can use GPS, START, GOALP, MOVES, MAKE, and PROGRESS CANNIBAL, for GPS, START,... FOX, with only a few slight changes: I must change (m m m c c c) to (fox goose corn) in starting conditions; MOVES must return (MOVE0) and (MOVE1) instead of (MOVE1) and (MOVE2), and

```
(use gps cannibal)
 (GPS CANNIBAL)

(use start cannibal but (replace m up1 with (fox goose
corn)))
 (START CANNIBAL)

(use goalp cannibal)
 (GOALP CANNIBAL)

(use moves cannibal but
(replace move1 with move0) (replace move2 with move1))
 (MOVES CANNIBAL)

(define move0 as list cons nil valueof from)
 (MOVE0)

(use make cannibal)
 (MAKE CANNIBAL)

(use progress cannibal)
 (PROGRESS CANNIBAL)
```

I must change the forbidden conditions. Instead of the countq advice, PROGRESS must check to see whether the goose is a member of the TO side. If anything else is also a member, it should quit.

```
(tell progress (instead of countq advice),
if ' goose is not a member of (valueof to), then ignore,
if (cdr valueof to) then quit)
 PROGRESS FOX)
```

Now GPS begins to solve the problem.

```
solve (fox, goose, and corn problem)
 BRING A FOX ACROSS
 SIDE1:  A GOOSE AND A CORN
 SIDE2:  A FOX


 BRING A GOOSE ACROSS
 SIDE1:  A FOX AND A CORINT. O
***  ERROR CALLED
```

I interrupt it because

```
(tell phrase after, if normal is equal to ' fox,
then return with (subst ' the ' a value))
 PHRASE
```

"THE FOX" sounds much better than "A FOX," and it's easy to change.

```
solve (fox, goose, and corn problem)
 BRING THE FOX ACROSS
 SIDE1: THE GOOSE AND THE CORN
 SIDE2:  THE FOX

 BRING THE GOOSE ACROSS
 SIDE1: THE FOX AND THE CORN
 SIDE2: THE GOOSE
```

```
BRING NOBODY BACK
SIDE1: THE FOX AND THE CORN
SIDE2: THE GOOSE

BRING THE FOX ACROSS
SIDE1: THE CORN
SIDE2: THE FOX AND THE GOOSE

BRING NOBODY BACK
SIDE1: THE CORN
SIDE2: THE FOX AND THE GOOSE

BRING THE FOX BACK
SIDE1: THE FOX AND THE CORN
SIDE2: THE GOOSE

BRING THE GOOSE BACK
SIDE1: THE GOOSE AND THE CORN
SIDE2: THE FOX

BRING THE GOOSE ACROSS
SIDE1: THE CORN
SIDE2: THE GOOSE AND THE FOX

BRING THE CORN ACROSS
SIDE1: THE GOOSE
SIDE2: THE CORN AND THE FOX

BRING NOBODY BACK
SIDE1: THE GOOSE
SIDE2: THE CORN AND THE FOX

BRING THE GOOSE ACROSS
SIDE1: NOBODY
SIDE2: THE GOOSE, THE CORN, AND THE FOX

(11 MOVES CONSIDERED)

*T*
```

The solution takes 7 moves. GPS only considers 11 moves all together.

```
solve (fox and goose)

(11 MOVES CONSIDERED)

 BRING THE GOOSE ACROSS, LEAVING THE FOX AND THE CORN
 ON SIDE1.

 BRING NOBODY BACK, LEAVING THE GOOSE ON SIDE2.

 BRING THE FOX ACROSS, LEAVING THE CORN ON SIDE1.

 BRING THE GOOSE BACK, LEAVING THE FOX ON SIDE2.
```

```
BRING THE CORN ACROSS, LEAVING THE GOOSE ON SIDE1.

BRING NOBODY BACK, LEAVING THE CORN AND THE FOX ON
SIDE2.

BRING THE GOOSE ACROSS, LEAVING NOBODY ON SIDE1.

*T*
```

I return to the cannibal and missionary problem and add (MOVE3) to the list of move types.

```
(change moves (insert (move3) after (move2)))
 (MOVES CANNIBAL)


(define move3 as alltran valueof from '($1 - $1 - $1)
'((2 4 6) 1 3 5 7))
 (MOVE3)
```

MOVE3 is defined in a fashion similar to MOVE2 and MOVE1 using ALLTRAN.

```
solve (cannibal and missionary problem)
  BRING THREE MISSIONARIES ACROSS
  SIDE1: THREE CANNIBALS
  SIDE2: THREE MISSIONARIES

  BRING A MISSIONARY BACK
  SIDE1: A MISSIONARY AND THREE CANNIBALS
  SIDE2: TWO MISSIONARIES

  BRING TWO MISSIONARIES BACK
  SIDE1: TWO MISSIONARIES AND THREE CANNIBALS
  SIDE2: A MISSIONARY

  BRING THREE MISSIONARIES BACK
  SIDE1: THREE MISSIONARIES AND THREE CANNIBALS
  SIDE2: NOBODY

  BRING TWO MISSIONARIES AND A CANNIBAL ACROSS
  SIDE1: A MISSIONARY AND TWO CANNIBALS
  SIDE2: TWO MISSIONARIES AND A CANNIBAL

  BRING A MISSIONARY AND TWO CANNIBALS ACROSS
  SIDE1: TWO MISSIONARIES AND A CANNIBAL
  SIDE2: A MISSIONARY AND TWO CANNIBALS
```

```
BRING THREE CANNIBALS ACROSS
SIDE1: THREE MISSIONARIES
SIDE2: THREE CANNIBALS

BRING A CANNIBAL BACK
SIDE1: A CANNIBAL AND THREE MISSIONARIES
SIDE2: TWO CANNIBALS

BRING A CANNIBAL AND TWO MISSIONARIES ACROSS
SIDE1: A MISSIONARY
SIDE2: THREE CANNIBALS AND TWO MISSIONARIES

BRING THREE MISSIONARIES ACROSS
SIDE1: A CANNIBAL
SIDE2: THREE MISSIONARIES AND TWO CANNIBALS

BRING A MISSIONARY BACK
SIDE1: A MISSIONARY AND A CANNIBAL
SIDE2: TWO MISSIONARIES AND TWO CANNIBALS

BRING A MISSIONARY AND A CANNIBAL ACROSS
SIDE1: NOBODY
SIDE2: THREE MISSIONARIES AND THREE CANNIBALS

(12 MOVES CONSIDERED)

BRING THREE CANNIBALS ACROSS, LEAVING THREE MISSIONARIES
ON SIDE1.

BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON SIDE2.

BRING THREE MISSIONARIES ACROSS, LEAVING A CANNIBAL ON
SIDE1.

BRING A MISSIONARY BACK, LEAVING TWO MISSIONARIES AND
TWO CANNIBALS ON SIDE2.

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
NOBODY ON SIDE1.

*T*
```

Now the solution only takes five moves instead of the eleven required for a two man boat. Only 12 moves are considered instead of 20. Both heuristics introduced earlier still operate in conjunction with this problem.

I would now like to be able to indicate the number of cannibal(s) or missionary(ies) in the initial statement of the problem. Preparatory to this, I'll have to make the PROGRESS evaluation a little more subtle.

```
(tell progress (instead of countq advice),
if (eaten side1) or (eaten side2) then quit)
 (PROGRESS CANNIBAL)
```

This replaces the old counting method with a call to the function EATEN.

```
(define eaten (x) as ' m is a member of x and
(eval cons ' plus x '((m . -1) (c . 1))) is greater than
0)
 (EATEN)
```

Basically what EATEN does is take the representation of a side, e.g., (m c m m c), puts "PLUS" in front of it, which yields (PLUS m c m m c), and evaluates this with m=-1 and c=1. If the resulting score is greater than zero, then the cannibals win, and the program must abandon this line of attack.

```
(change start (replace m up2 with (flipq (sublis
'((cannibals . c) (cannibal . c) (missionaries . m)
(missionary . m)) solve)
((either (- for $1 $1 and $1 $1) (-)))
((either ((repeat (= car 3) (/t 1 4)) (repeat (= car -2)
(/t 1 -1)))
     (m m m c c c))))))
 (START CANNIBAL)
```

START must be modified accordingly. When I say SOLVE (... FOR N Y AND M Z), START will make a list of N Y's and M Z's be the starting conditions; otherwise it uses (m m m c c c).

I give GPS an easy problem to check it out.

```
solve (cannibal and missionary problem for 1 cannibal
and 1 missionary)
 BRING A CANNIBAL AND A MISSIONARY ACROSS
 SIDE1:  NOBODY
 SIDE2: A CANNIBAL AND A MISSIONARY

(1 MOVES CONSIDERED)
```

```
BRING A CANNIBAL AND A MISSIONARY ACROSS, LEAVING
NOBODY ON SIDE1.

*T*
```

Now I am going to make it say (1 MOVE CONSIDERED). First I'll define a function AGREE, which does what a portion of PHRASE used to do.

```
(define agree (x y) as list sublis '((1 . 1) (2 . two)
(3 . three)) x, if x is equal to 1 then y, else
(plural y))
 (AGREE)


agree (1 cannibal)
 (1 CANNIBAL)
agree (2 missionary)
 (TWO MISSIONARIES)
```

This is what AGREE is supposed to do.

```
(change start after (replace printred n1 with
(nconc agree number ' move '(considered))))
 (START AFTER)
```

Now I change START.

```
solve (cannibal and missionary problem for 1 cannibal
and 1 missionary)
 BRING A CANNIBAL AND A MISSIONARY ACROSS
 SIDE1: NOBODY
 SIDE2: A CANNIBAL AND A MISSIONARY

(1 MOVE CONSIDERED)

 BRING A CANNIBAL AND A MISSIONARY ACROSS, LEAVING
 NOBODY ON SIDE1.

*T*
```

Now I try the problem with 4 cannibals and 4 missionaries. This can't be solved with only a two man boat.

```
solve (cannibal and missionaries for 4 cannibals and
4 missionaries)

 (72 MOVES CONSIDERED)

 NIL
```

I read in the advice for the three-man boat that was made earlier. (PILOT had saved it under the file GPS7 LISP.)

```
evalread (gps7 lisp speak)
 (CHANGE MOVES (INSERT (MOVE3) AFTER (MOVE2)))
 (MOVES CANNIBAL)
 (DEFINE MOVE3 AS ALLTRAN VALUEOF FROM ' ($1 $ $1 $ $1)
 ' ((2 4 6) 1 3 5 7))
 (MOVE3)
 STOP


solve (cannibal and missionaries for 4 cannibals and
4 missionaries)

 (17 MOVES CONSIDERED)

 BRING THREE CANNIBALS ACROSS, LEAVING A CANNIBAL AND
 4 MISSIONARIES ON SIDE1.

 BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON SIDE2.

 BRING TWO CANNIBALS ACROSS, LEAVING 4 MISSIONARIES ON
 SIDE1.

 BRING A CANNIBAL BACK, LEAVING THREE CANNIBALS ON
 SIDE2.

 BRING THREE MISSIONARIES ACROSS, LEAVING A CANNIBAL
 AND A MISSIONARY ON SIDE1.

 BRING A MISSIONARY AND A CANNIBAL BACK, LEAVING TWO
 MISSIONARIES AND TWO CANNIBALS ON SIDE2.

 BRING TWO MISSIONARIES AND A CANNIBAL ACROSS, LEAVING
 A CANNIBAL ON SIDE1.

 BRING A MISSIONARY BACK, LEAVING THREE CANNIBALS AND
 THREE MISSIONARIES ON SIDE2.

 BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
 NOBODY ON SIDE1.

*T*
```

Now GPS solves this problem and another one.

```
solve (cannibal and missionaries for 4 cannibals and
5 missionaries)

 (10 MOVES CONSIDERED)

  BRING THREE CANNIBALS ACROSS, LEAVING A CANNIBAL AND
  5 MISSIONARIES ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON SIDE2.

  BRING THREE MISSIONARIES ACROSS, LEAVING TWO CANNIBALS
  AND TWO MISSIONARIES ON SIDE1.

  BRING A MISSIONARY BACK, LEAVING TWO MISSIONARIES AND
  TWO CANNIBALS ON SIDE2.

  BRING TWO MISSIONARIES AND A CANNIBAL ACROSS, LEAVING
  A CANNIBAL AND A MISSIONARY ON SIDE1.

  BRING A MISSIONARY BACK, LEAVING THREE CANNIBALS AND
  THREE MISSIONARIES ON SIDE2.

  BRING TWO MISSIONARIES AND A CANNIBAL ACROSS, LEAVING
  NOBODY ON SIDE1.

 *T*


 (change moves (delete move3 up1))
   (MOVES CANNIBAL)
```

I would like to solve this latter problem with the original two man boat so I delete the MOVE3 advice.

```
solve (cannibal and missionaries for 4 cannibals and
5 missionaries)

 (30 MOVES CONSIDERED)

  BRING TWO CANNIBALS ACROSS, LEAVING TWO CANNIBALS
  AND 5 MISSIONARIES ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING A CANNIBAL ON SIDE2.

  BRING TWO CANNIBALS ACROSS, LEAVING A CANNIBAL AND
  5 MISSIONARIES ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON
  SIDE2.

  BRING TWO MISSIONARIES ACROSS, LEAVING TWO CANNIBALS
  AND THREE MISSIONARIES ON SIDE1.
```

```
BRING A CANNIBAL BACK, LEAVING TWO MISSIONARIES AND
A CANNIBAL ON SIDE2.

BRING A CANNIBAL AND A MISSIONARY ACROSS, LEAVING TWO
CANNIBALS AND TWO MISSIONARIES ON SIDE1.

BRING A MISSIONARY BACK, LEAVING TWO CANNIBALS AND TWO
MISSIONARIES ON SIDE2.

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING A
CANNIBAL AND TWO MISSIONARIES ON SIDE1.

BRING A CANNIBAL BACK, LEAVING THREE MISSIONARIES
AND TWO CANNIBALS ON SIDE2.

BRING A CANNIBAL AND A MISSIONARY ACROSS, LEAVING A
CANNIBAL AND A MISSIONARY ON SIDE1.

BRING A MISSIONARY BACK, LEAVING THREE CANNIBALS AND
THREE MISSIONARIES ON SIDE2.

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING A
MISSIONARY ON SIDE1.

BRING A CANNIBAL BACK, LEAVING 4 MISSIONARIES AND
THREE CANNIBALS ON SIDE2.

BRING A CANNIBAL AND A MISSIONARY ACROSS, LEAVING
NOBODY ON SIDE1.

*T*
```

This solution takes 15 moves, as opposed to a seven move solution for a larger boat. In the latter case, 10 moves were considered, while in this one 30 moves were considered.

Rather than continually adding and removing the MOVE3 advice, I would like to specify the size of the boat as an input parameter.

```
(start : (put (list (flipq solve ((either (- for $1
($set num (= car -1)) in a boat -) (($set num '2) -)))
 (advice quote (((repeat (= num) (moven (= (car n
 /// list ' moves ' cannibal))))))))))))
 START
```

If I say "a boat that can carry 3," MOVES will return with ((MOVEN 1), (MOVEN 2), (MOVEN 3)).

MOVEN is similar to MOVE1, MOVE2, and MOVE3; it computes the appropriate pattern and format for ALLTRAN. If n is 3, it constructs the list (1 2 3 4 5 6 7), and uses this to assemble the pattern ($1 $ $1 $ $1) and the format ((2 4 6) 1 3 5 7) for ALLTRAN. Note that these are identical to those in the original definition of MOVE3.

```
(define moven (n) as prog2 setq n flipq thrun add1
   (times 2 n) ((repeat $2 $1) $1)  (((repeat $ $1))
   (repeat 2) (repeat 1) -1), alltran valueof from cdar
   n cdr n)

***  ERROR CALLED
(PARSING : PROG2 SETQ N FLIPQ THRUN ADD1 (TIMES 2 N)
((REPEAT $1 $1) $1) (((REPEAT $ $1)) (REPEAT 2)
(REPEAT 1) -1) ALLTRAN VALUEOF FROM CDAR N CDR N)

(EDIT OR FORGET IT)
```

This error is because THRUN has not been defined at this point. Therefore, the parsing routine doesn't know how many arguments is has. (THRUN will be the function that constructs the list (1 2 3 4 5 6 7).) I do this section of parsing myself by substituting in (THRUN (ADD1 (TIMES 2 N))) for the unparsed segment.

```
(replace thrun n2 with (thrun (add1 (times 2 n))))
***  ERROR CALLED
 (PARSING : PROG2 SETQ N FLIPQ THRUN (THRUN (ADD1
 (TIMES 2 N))) ((REPEAT $1 $1) $1) (((REPEAT $ $1))
 (REPEAT 2) (REPEAT 1) -1) ALLTRAN VALUEOF FROM
 CDAR N CDR N)

 (EDIT OR FORGET IT)
```

I forgot to take out THRUN.

```
edit
(delete thrun)
(insert (quote -) after repeat - repeat)
x
 (DEFINE MOVEN (N) AS PROG2 SETQ N FLIPQ (THRUN (ADD1
 (TIMES 2 N))) ((REPEAT $1 $1) $1) (((REPEAT -- $ $1))
 (REPEAT 2) (REPEAT 1) -1) ALLTRAN VALUEOF FROM CDAR
 N CDR N)
stop
 (MOVEN)

(define thrun (n) as prog (x)
loop setq x cons n x,
if (zerop setq n sub1 n) then (return x) end,
go loop)
 (THRUN)

thrun (7)

 (1 2 3 4 5 6 7)
```

Now I define THRUN

```
breaklist (moves moven)
 (MOVES MOVEN)
```

and BREAK on MOVES and MOVEN to see if they are correct.

```
solve (cannibal and missionary problem for 3 in a boat
and for 4 cannibals and 5 missionaries)
 (BREAK IN MOVES)
```

I get a BREAK in MOVES,

```
stop
 (VALUE OF MOVES)
((MOVEN 3) (MOVEN 2) (MOVEN 1))
```

with the correct value. Note that it is reversed because of the heuristic introduced earlier.

```
 (BREAK IN MOVEN)
n
 3
```

A Break in MOVEN; I ask for the value of N; it is 3. I ask that MOVEN be evaluated.

```
eval
 GC AT 03041 FULL WORDS  723 FREE   148  PUSH DOWN
 DEPTH   270
*** ERROR  NOROOM
 NIL
  (BREAK IN MOVEN)
```

The BREAK is maintained in spite of the error. I wipe out the EDIT routines to make space, and go on.

```
(wipe edit)
 (EDIT)
eval
*** ERROR  NUMVAL
 (($ $1 $ $1 $ $1) 2 4 6 1 3 5 7)
 (BREAK IN MOVEN)
```

This error is because N has been changed by MOVEN, I must reset it to 3, which I do.

```
(setq n 3) eval
 3
 (MOVEN EVALUATED)
(car moven)
 (C C C C M M M M M)
(cadddr moven)
 (C C M C C M M M M)
```

MOVEN is evaluated, I look at the first element of its value, and at the third element - both are wrong. I BREAK ALLTRAN, reset n and try again.

```
(breaklist alltran)
 (ALLTRAN)
(setq n 3)
 3
eval
 (BREAK IN ALLTRAN)
z
 (2 4 6 1 3 5 7)
```

ALLTRAN is not getting the right value for z. I'll set it correctly and see if anything else is wrong.

```
(setq z '((2 4 6) 1 3 5 7))
 ((2 4 6) 1 3 5 7)
eval
 GC AT 03041  FULL WORDS  730 FREE    106  PUSH DOWN
 DEPTH  361
*** ERROR NOROOM
 NIL
 (BREAK IN ALLTRAN)
```

I ran out of space again. This time I wipe the SYSTEM routines.

```
(wipe '' system)
 (SYSTEM)
eval
 (ALLTRAN EVALUATED)
(car alltran)
 ((C C C) C M M M M M)
(cadddr alltran)
 ((C C M) C C M M M M)
```

ALLTRAN is correct. I quit, and go back to the top, and restore SYSTEM and EDIT.

```
ok
 (ALLTRAN)
 (MOVEN EVALUATED)
quit
***  ERROR  CALLED
 (MOVEN)
restore (system edit)
 (SYSTEM EDIT)
```

The first thing to do is correct the bug in MOVEN.

```
(change moven expr (replace '(repeat 2) with ((repeat
2))))
 (MOVEN EXPR)
```

```
(before all : if fsleft is less than 500, then (makeroom))
 ALL
```

I decide to have the system itself make room, I can do this by advising ALL functions to check the number of words left.

```
(define makeroom as prog (x)
setq x '(update edit system break),
fancyprint cons lastfn append '(: only) cons fsleft
   append '(words left) (list period cr),
loop     if (get car x ' wiped) is null  then
   (fancyprint append ' (i had to wipe) cons wipe1 car x
     list period cr),
if pop x then (go loop)    )
 (MAKEROOM)
```

MAKEROOM calls WIPE on (UPDATE EDIT SYSTEM BREAK) until it can find something to wipe out, and then prints an appropriate message.

```
solve (cannibal and missionary problem for 3 in a boat
and for 4 cannibals and 5 missionaries)
 (BREAK IN MOVES)
(unbreaklist '' moves)
 (MOVES)
ok
 (MOVES)
 (BREAK IN MOVEN)
eval
 (BREAK IN ALLTRAN)
eval
 GC AT 03041  FULL WORDS  715 FREE   140  PUSH DOWN
 DEPTH   450
***  ERROR  NOROOM
 NIL
 (BREAK IN ALLTRAN)
```

The system didn't call MAKEROOM because it ran out of space while inside of a function that is not advised, namely AM1, a subfunction of ALLTRAN. If I give AM1 some advice, then the check for available space will also be performed here.

```
(system)
(tell am1 before nil)
  SYS1:  ONLY 994 WORDS LEFT.
  I HAD TO WIPE EDIT.
 AM1

ok
 NIL
```

While advising AM1, SYS1 ran into a situation in which there were fewer than 994 words left - actually there were only 140 according to the error message. However, a garbage collection occurred before the print out of the message and so it states, somewhat contradictorily, that there are only 994 words left.

```
eval
```

I go on with the GPS problem.

```
  AM1:  ONLY 453 WORDS LEFT.
  I HAD TO WIPE SYSTEM
 (ALLTRAN EVALUATED)
(car alltran)
 ((C C C) C M M M M M)
```

Correct.

```
(unbreaklist alltran moven)
 (ALLTRAN MOVEN)
```

Unbreak everything and go.

```
ok
 (ALLTRAN)
 (MOVEN EVALUATED)
ok
 (MOVEN)
  BRING THREE CANNIBALS ACROSS
  SIDE1: A CANNIBAL AND 5 MISSIONARIES
  SIDE2: THREE CANNIBALS
```

```
 BRING A CANNIBAL BACK
 SIDE1: TWO CANNIBALS AND 5 MISSIONARIES
 SIDE2: TWO CANNIBALS

 BRING TWO CANNIBALS AND A MISSIONARY ACROSS
 SIDE1: 4 MISSIONARIES
 SIDE2: 4 CANNIBALS AND A MISSIONARY

 BRING A CANNIBAL AND TWO MISSIONARIES ACROSS
 SIDE1: A CANNIBAL AND THREE MISSIONARIES
 SIDE2: THREE CANNIBALS AND TWO MISSIONARIES

 BRING THREE MISSIONARIES ACROSS
 SIDE1: TWO CANNIBALS AND TWO MISSIONARIES
 SIDE2: THREE MISSIONARIES AND TWO CANNIBALS

 BRING A MISSIONARY BACK
 SIDE1: THREE MISSIONARIES AND TWO CANNIBALS
 SIDE2: TWO MISSIONARIES AND TWO CANNIBALS

 BRING A MISSIONARY AND TWO CANNIBALS ACROSS
 SIDE1: TWO MISSIONARIES
 SIDE2: THREE MISSIONARIES AND 4 CANNIBALS

 BRING TWO MISSIONARIES AND A CANNIBAL ACROSS
 SIDE1: A CANNIBAL AND A MISSIONARY
 SIDE2: 4 MISSIONARIES AND THREE CANNIBALS

 BRING A MISSIONARY BACK
 SIDE1: TWO MISSIONARIES AND A CANNIBAL
 SIDE2: THREE CANNIBALS AND THREE MISSIONARIES

 BRING TWO MISSIONARIES AND A CANNIBAL ACROSS
 SIDE1: NOBODY
 SIDE2: 5 MISSIONARIES AND 4 CANNIBALS

(10 MOVES CONSIDERED)

 BRING THREE CANNIBALS ACROSS, LEAVING A CANNIBAL AND
 5 MISSIONARIES ON SIDE1.

 BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON SIDE2.

 BRING THREE MISSIONARIES ACROSS, LEAVING TWO CANNIBALS
 AND TWO MISSIONARIES ON SIDE1.

 BRING A MISSIONARY BACK, LEAVING TWO MISSIONARIES AND
 TWO CANNIBALS ON SIDE2.

 BRING TWO MISSIONARIES AND A CANNIBAL ACROSS, LEAVING
 A CANNIBAL AND A MISSIONARY ON SIDE1.

 BRING A MISSIONARY BACK, LEAVING THREE CANNIBALS AND
 THREE MISSIONARIES ON SIDE2.

 BRING TWO MISSIONARIES AND A CANNIBAL ACROSS, LEAVING
 NOBODY ON SIDE1.

*T*
```

I now solve various problems.

```
solve (cannibal)

 (20 MOVES CONSIDERED)

 *T*
solve (cannibal and missionaries for 3 in a boat)

 (12 MOVES CONSIDERED)

  BRING THREE CANNIBALS ACROSS, LEAVING THREE
  MISSIONARIES ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON
  SIDE2.

  BRING THREE MISSIONARIES ACROSS, LEAVING A CANNIBAL
  ON SIDE1.

  BRING A MISSIONARY BACK, LEAVING TWO MISSIONARIES
  AND TWO CANNIBALS ON SIDE2.

  BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
  NOBODY ON SIDE1.

 *T*
solve (cannibal and missionaries for 3 in a boat and
for 4 cannibals and 4 missionaries)
  AM1:  ONLY 463 WORDS LEFT.
  I HAD TO WIPE EDIT.

 (17 MOVES CONSIDERED)

  BRING THREE CANNIBALS ACROSS, LEAVING A CANNIBAL
  AND 4 MISSIONARIES ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON
  SIDE2.

  BRING TWO CANNIBALS ACROSS, LEAVING 4 MISSIONARIES
  ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING THREE CANNIBALS ON
  SIDE2.

  BRING THREE MISSIONARIES ACROSS, LEAVING A CANNIBAL
  AND A MISSIONARY ON SIDE1.

  BRING A MISSIONARY AND A CANNIBAL BACK, LEAVING TWO
  MISSIONARIES AND TWO CANNIBALS ON SIDE2.

  BRING TWO MISSIONARIES AND A CANNIBAL ACROSS, LEAVING
  A CANNIBAL ON SIDE1.
```

```
BRING A MISSIONARY BACK, LEAVING THREE CANNIBALS
AND THREE MISSIONARIES ON SIDE2.

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
NOBODY ON SIDE1.

*T*
```

Now I try a new problem - which GPS can't solve.

```
solve (how big a boat do you need for 4 cannibals
  and 4 missionaries)
 (DONT KNOW HOW)

(start : bind conditions to (flipq solve (- for -)
  (solve cannibal and missionary for n in a boat and
  for -)))
 START
```

If I say (HOW BIG A BOAT DOES IT TAKE FOR ...), CONDITIONS will be bound to (SOLVE CANNIBAL AND MISSIONARY FOR N IN A BOAT AND FOR ...).

```
(tell start, return with (prog (n)
setq n 1,
loop  if (valueof subst n ' n conditions) then
   (return append '(a boat that can carry) list n) end,
increment n, go loop))
 START
```

This advice will cause START to loop, calling SOLVE for different values of N. Now GPS can solve the problem.

```
solve (how big a boat do you need for 4 cannibals and
  4 missionaries) '

 (THREE MOVES CONSIDERED)

  MAKE: ONLY 382 WORDS LEFT.
  I HAD TO WIPE EDIT.

 (72 MOVES CONSIDERED)


 (17 MOVES CONSIDERED)
```

```
BRING THREE CANNIBALS ACROSS, LEAVING A CANNIBAL
AND 4 MISSIONARIES ON SIDE1.

BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON
SIDE2.

BRING TWO CANNIBALS ACROSS, LEAVING 4 MISSIONARIES
ON SIDE1.

BRING A CANNIBAL BACK, LEAVING THREE CANNIBALS ON
SIDE2.

BRING THREE MISSIONARIES ACROSS, LEAVING A CANNIBAL
AND A MISSIONARY ON SIDE1.

BRING A MISSIONARY AND A CANNIBAL BACK, LEAVING TWO
MISSIONARIES AND TWO CANNIBALS ON SIDE2.

BRING TWO MISSIONARIES AND A CANNIBAL ACROSS,
LEAVING A CANNIBAL ON SIDE1.

BRING A MISSIONARY BACK, LEAVING THREE CANNIBALS
AND THREE MISSIONARIES ON SIDE2.

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
NOBODY ON SIDE1.

(A BOAT THAT CAN CARRY 3)
```

GPS considered three moves with a boat that could only carry 1, 72 moves with a boat that could carry 2, and found the answer with a boat that can carry 3.

Now I am going to introduce a new supermissionary - a tarzan, who cannot be eaten, although he can help to outnumber the cannibals and protect the missionaries, and can also row the boat across.

```
(change eaten expr (insert (x . -1) after (m . -1)))
 (EATEN EXPR)
```

I use X to stand for the new element.

```
(change start
(insert (tarzan . x) (tarzans . x) after (missionary
  . m))
(replace either n1 with (- for (repeat 2 $1 /
  (numberp) $1) -))
(replace either - either n1 with
  ((repeat m (repeat (= car 1) (/r m 2))))))
 (START CANNIBAL)
```

Instead of saying SOLVE (CANNIBAL FOR N MISSIONARIES AND N CANNIBALS) I now say SOLVE (CANNIBAL FOR N MISSIONARIES M CANNIBALS P TARZANS). Actually this advice modification to START will allow it to handle any number of different types of people.

```
(change start after (insert (x . tarzan) after
  (m . missionary )))
 (START AFTER)

(change make (insert (x . tarzan) after (m .
  missionary )))
 (MAKE CANNIBAL)
```

Now I try it out. Note that since I don't tell it how big a boat to use, GPS assumes a two man boat.

```
solve (cannibal and missionary problem for 3
  cannibals, 2 missionaries, 1 tarzan)
 BRING TWO CANNIBALS ACROSS
 SIDE1: A CANNIBAL, TWO MISSIONARIES, AND A TARZAN
 SIDE2: TWO CANNIBALS

 BRING A CANNIBAL BACK
 SIDE1: TWO CANNIBALS, TWO MISSIONARIES, AND A TARZAN
 SIDE2: A CANNIBAL

 BRING TWO CANNIBALS ACROSS
 SIDE1: TWO MISSIONARIES AND A TARZAN
 SIDE2: THREE CANNIBALS

 BRING A CANNIBAL BACK
 SIDE1: A CANNIBAL, TWO MISSIONARIES, AND A TARZAN
 SIDE2: TWO CANNIBALS

 BRING A CANNIBAL AND A MISSIONARY ACROSS
 SIDE1: A MISSIONARY AND A TARZAN
 SIDE2: THREE CANNIBALS AND A MISSIONARY
```

```
BRING A CANNIBAL AND A TARZAN ACROSS
SIDE1: TWO MISSIONARIES
SIDE2: THREE CANNIBALS AND A TARZAN

BRING A CANNIBAL BACK
SIDE1: A CANNIBAL AND TWO MISSIONARIES
SIDE2: A TARZAN AND TWO CANNIBALS

BRING A CANNIBAL AND A MISSIONARY ACROSS
SIDE1: A MISSIONARY
SIDE2: THREE CANNIBALS, A MISSIONARY, AND A TARZAN

BRING TWO MISSIONARIES ACROSS
SIDE1: A CANNIBAL
SIDE2: TWO MISSIONARIES, A TARZAN, AND TWO CANNIBALS

BRING A MISSIONARY BACK
SIDE1: A MISSIONARY AND A CANNIBAL
SIDE2: A MISSIONARY, A TARZAN, AND TWO CANNIBALS

BRING A MISSIONARY AND A CANNIBAL ACROSS
SIDE1: NOBODY
SIDE2: TWO MISSIONARIES, A TARZAN, AND THREE
       CANNIBALS

(11  MOVES CONSIDERED)

AGREE: ONLY 493 WORDS LEFT.
I HAD TO WIPE EDIT.
BRING TWO CANNIBALS ACROSS, LEAVING A CANNIBAL, TWO
MISSIONARIES, AND A TARZAN ON SIDE1.

BRING A CANNIBAL BACK, LEAVING A CANNIBAL ON SIDE2.

BRING TWO CANNIBALS ACROSS, LEAVING TWO MISSIONARIES
AND A TARZAN ON SIDE1.

BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON
SIDE2.

BRING A CANNIBAL AND A TARZAN ACROSS, LEAVING TWO
MISSIONARIES ON SIDE1.

BRING A CANNIBAL BACK, LEAVING A TARZAN AND TWO
CANNIBALS ON SIDE2.

BRING TWO MISSIONARIES ACROSS, LEAVING A CANNIBAL
ON SIDE1.

BRING A MISSIONARY BACK, LEAVING A MISSIONARY, A
TARZAN, AND TWO CANNIBALS ON SIDE2.

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
NOBODY ON SIDE1.

*T*
```

The solution is only nine moves long, the minimum to transfer six people, as opposed to the eleven without Tarzan's help.

Now I give it a trivial problem - nobody can get eaten,

```
solve (cannibal and missionaries for 3 cannibals,
3 tarzans)

 (9 MOVES CONSIDERED)

  BRING TWO CANNIBALS ACROSS, LEAVING A CANNIBAL AND
  THREE TARZANS ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING A CANNIBAL ON SIDE2.

  BRING TWO CANNIBALS ACROSS, LEAVING THREE TARZANS
  ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING TWO CANNIBALS ON
  SIDE2.

  BRING A CANNIBAL AND A TARZAN ACROSS, LEAVING TWO
  TARZANS ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING A TARZAN AND TWO
  CANNIBALS ON SIDE2.

  BRING A CANNIBAL AND A TARZAN ACROSS, LEAVING A
  TARZAN ON SIDE1.

  BRING A CANNIBAL BACK, LEAVING TWO TARZANS AND TWO
  CANNIBALS ON SIDE2.

  BRING A CANNIBAL AND A TARZAN ACROSS, LEAVING
  NOBODY ON SIDE1.

 *T*
```

and this problem combining all of the things I have told the problem solver.

```
solve (how big a boat do you need for 3 missionaries,
  1 tarzan, 4 cannibals)

 (4 MOVES CONSIDERED)

 (30 MOVES CONSIDERED)
```

BRING A MISSIONARY AND A CANNIBAL ACROSS, LEAVING
TWO MISSIONARIES, A TARZAN, AND THREE CANNIBALS
ON SIDE1.

BRING A MISSIONARY BACK, LEAVING A CANNIBAL ON SIDE2.

BRING A TARZAN AND A CANNIBAL ACROSS, LEAVING THREE
MISSIONARIES AND TWO CANNIBALS ON SIDE1.

BRING A TARZAN BACK, LEAVING TWO CANNIBALS ON SIDE2.

BRING A TARZAN AND A MISSIONARY ACROSS, LEAVING TWO
MISSIONARIES AND TWO CANNIBALS ON SIDE1.

BRING A MISSIONARY BACK, LEAVING A TARZAN AND TWO
CANNIBALS ON SIDE2.

BRING TWO CANNIBALS ACROSS, LEAVING THREE MISSIONARIES
ON SIDE1.

BRING A CANNIBAL BACK, LEAVING THREE CANNIBALS AND
A TARZAN ON SIDE2.

BRING TWO MISSIONARIES ACROSS, LEAVING A CANNIBAL
AND A MISSIONARY ON SIDE1.

BRING A MISSIONARY AND A CANNIBAL BACK, LEAVING A
MISSIONARY, A TARZAN, AND TWO CANNIBALS ON SIDE2.

BRING TWO MISSIONARIES ACROSS, LEAVING TWO CANNIBALS
ON SIDE1.

BRING A TARZAN BACK, LEAVING THREE MISSIONARIES
AND TWO CANNIBALS ON SIDE2.

BRING A TARZAN AND A CANNIBAL ACROSS, LEAVING A
CANNIBAL ON SIDE1.

BRING A TARZAN BACK, LEAVING THREE CANNIBALS AND
THREE MISSIONARIES ON SIDE2.

BRING A TARZAN AND A CANNIBAL ACROSS, LEAVING NOBODY
ON SIDE1.

(A BOAT THAT CAN CARRY 2)

This empty page was substituted for a blank page in the original document.

CHAPTER 7

IMPROVING PILOT

PILOT is the result of an evolutionary process extending over more than two years. However, there is no reason to assume that this process has terminated, nor that PILOT has reached some sort of ultimate state. This chapter discusses ways that PILOT might be improved at several levels, ranging from specific suggestions for modifying the function ADVISE, through changes in FLIP and the translator, to extensions of the subjective model for programming. However, the significance of PILOT lies not in any specific characteristics or features it possesses, but rather in that it demonstrates that it is possible to get computers to participate in, and cooperate with, research efforts in programming to a much greater extent than is now being done.

Questions of Efficiency

The heart of the PILOT system is the function ADVISE, which executes a procedure along with its advice. In attempting to evaluate PILOT's efficiency, we must compare programs written using PILOT, i.e., in which ADVISE is called to interpret procedures, with those written directly in LISP. The same program written in machine language would probably be more efficient. But, there is always a tradeoff in efficiency between generality and specificity. Presumably the ease of programming compensates for this factor, or you would not use the more general system.

The question, therefore, is: assuming your program is to be written in LISP, how much does it cost you to do it within PILOT, i.e., using ADVISE? This will then have to be weighed against the conveniences of being able to make changes immediately by advising.

If the program is to be run interpretively, as opposed to compiled, the cost is practically zero. This is because ADVISE and its satellite functions are all compiled. Therefore, the overhead involved in calling ADVISE is small compared with the time required to interpret the pieces of advice. These would have to be interpreted in some form anyway, either as advice, or as a part of the definition of the function. For example, suppose you wish to modify the function PROGRESS in the example in Chapter 6 so that whenever the cannibals would eat the missionaries, PROGRESS returns NIL. Then somewhere, either in the definition of PROGRESS or as a piece of advice, there must be some S-expression representation of this computation, in the form of a conditional with appropriate clauses. This conditional must at some point be interpreted for PROGRESS to work as intended. If PROGRESS is uncompiled, the difference between interpreting this modification as advice, and including it in the function definition directly, is small.

For completeness, I include here computation times* for some of the experiments in Chapters 5 and 6. These are for programs run interpretively, using ADVISE. These figures do not include time spent in garbage collection.

Remarks	Time (seconds)
[Chapter 5: Deductive Question-Answering System]	
Question: (AT PENCIL COUNTY)	
no modifications	18.2
questions not containing variables are answered only once	14.2
with English output	27.0

* Although PILOT operates in a time-shared environment, these times are actual CPU times as computed by interrogating an internal clock.

Remarks	Time (seconds)
Question: (AT PENCIL Y) - corpus permits endless deduction	
limit on number of recursive calls to SOLUTION1 set at 1; answer (AT PENCIL DESK); 3 questions considered	3.5
limit set at 2; answer ((AT PENCIL DESK) (AT PENCIL HOME)): 11 questions considered	11.5
limit set at 3, answer ((AT PENCIL DESK) (AT PENCIL HOME) (AT PENCIL COUNTY): 39 questions	41.7
no limit - if question is repeated, return all answers found so far; 17 questions considered	14.2
[Chapter 6: General Problem Solver]	
Problem: Cannibal and Missionary	
No heuristic, 68 moves	14.5
Heuristic: bring two across, one back; 35 moves	8.9
Heuristic: do not attempt moves considered previously; 20 moves	7.5

If the user wishes to compile his programs, the question of efficiency becomes more serious. Although each individual piece of advice can be compiled, the overhead involved in calling ADVISE is now proportionally larger. It might even be desirable to include in PILOT a feature for collapsing advice into the function definition prior to compilation, so that it would then run as one compiled subroutine, without calling ADVISE. However, if it became necessary to make modifications subsequent to compilation, the user must either revert to calling ADVISE with the function, or else save its symbolic definition and recompile. In addition, eliminating the call to ADVISE means that HISTORY would not record an entry for this function.

The question here is basically one of open subroutines versus closed subroutines. The principal advantage of using closed subroutines for making modifications, as implemented with

ADVISE, is that it is easy to locate individual pieces of advice, and to change them, perhaps even by advising. It is also easier to continue making modifications after the function is compiled. This may be outweighed by considerations of speed. Probably both options should be included in future systems - provided that space is not an important factor. The user could then allow the particulars of the situation dictate his choice on which method to use.

* * *

This entire discussion has compared the efficiency of interface modifications performed with ADVISE with those performed editing the LISP function. There are, modifications which do not properly fall under the heading of interface modifications, even though they could be performed that way. For example, suppose FOO is a function of two arguments X and Y, and it is discovered that the order of these arguments has been reversed in the definition of FOO. It would be possible to correct this by advising: exchange X and Y before FOO was entered. Obviously this is much less efficient than correcting the function definition. The previous discussion compares the advice method with editing FOO by inserting a computation which exchanged X and Y, and not with reversing the order of the arguments in the definition. Comparing advising with the optimal method of modifying would bring us into a discussion of what is the most efficient program for a particular task. I am not prepared to discuss this latter question.

Improving FLIP

FLIP is also the result of an evolutionary process. Since it forms the basis for the translating and editing functions, and is also used by the programmer directly, it is worthwhile

to concentrate efforts on improving it. In particular, two additional semantic features in FLIP would be most useful. These are the multiple workspace and the depth search pattern.

Multiple Workspace

In most pattern-driven languages, the user matches a piece of data against a pattern. However, occasionally you want to match a piece of data against another piece of data, according to some pattern. For example, A matches B if whenever A is of the form (x y z ...), B is of the form ($1 x $1 y $1 z ...). Determining a match of this type involves a back and forth process that cannot easily be expressed except in programs written specifically for this purpose.

More generally, suppose it is necessary to process two lists using FLIP-type of operations, where the processing must go on simultaneously because the processing of one list affects the other. For example, suppose you wanted to find the longest common substring of two strings. This type of problem can best be solved by allowing two workspaces, instead of only one.

Some syntactic and semantic problems remain to be solved. The user must be able to indicate under what conditions to abandon processing one list and go to the other - since nothing can really occur simultaneously. It may also be necessary to specify more than one pattern.

Depth Search Pattern

When the user writes ($ A $ D $), he intends to find the first A followed by the first D, regardless of where it appears. In COMIT, this presents no difficulty because everything is at

the same level. However, in LISP this pattern will not match with the list (X Y Z (E T A I O N) X Y Z (S H R D L U) X Y Z). To match with this list, one must use the pattern ($ ($ A $) $ ($ D $) $). However, this latter pattern will not match with the first list. How can the user specify a match that is to occur at any depth?

This problem is of obvious importance in searching list structures. The user may not know at what depth a particular structure occurs, even though he may be able to specify a transformation on it. The depth search pattern would allow him to write ($$ $1 2 2 $$) to search for three repeated elements, at any depth as indicated by the "$$". The format (1 2 -1) would then transform the structure, deleting the two repetitions.

Improving the Language Syntax

One obvious place to improve PILOT is in the translator. This device is a collection of transformations, each of which is irrevocable, each of which operates with no information concerning the others. Often, a translation will succeed or fail depending on the chance order to which two transformations are applied. In the current translator, this situation is avoided by having the user segment parts of the input string with parentheses whenever there is a danger of misinterpretation. However, this quickly becomes cumbersome. Moreover, it places the burden on the user, instead of on the system, where it should be.

Bobrow[3,4] has shown that in a limited semantic context, that of algebra story problems, it is possible to relax syntactic conventions considerably. The input to his STUDENT program is

in the form of natural language, which the program "understands" in the context of algebra story problems.

Since the inputs to PILOT represent computations, it should similarly be possible to relax the syntactic restrictions. If an input string does not parse, i.e., if it does not translate into a recognizable computation - in our case a LISP function with its arguments - then clearly something is wrong. Somewhere a transformation was applied that should not have been. Before the system complains, we should have it back up and "undo" some of the transformations it executed. By this simple device, many ambiguities could be resolved.

For example, consider the input (TELL FOO TO INCREMENT X AND (PRINT Y)). The user intends this piece of advice to consist of two operations: incrementing the variable x and printing the value of y. However, this will translate into (TELL FOO TO INCREMENT (AND X (PRINT Y))), at which point the system complains. This is because the AND transformation, in the sense of (A AND B OR C), operated before the INCREMENT transformation. This AND, however, is intended to be the AND in the (TO ... AND ... AND ...) transformation. But, it is not recognized because INCREMENT has not yet operated.

Of course, this situation could be rectified by having INCREMENT operate first, perhaps by establishing a precedence on transformations. However, as the number of transformations used in the translator increases, the number of words used in two or more different contexts, e.g., AND, will also increase. Unless the user is constrained to writing AND1 and AND2 to indicate the two meanings of AND, some device for tentatively trying a

transformation becomes a necessity.

Extending the Language Semantics

Programming languages are designed to allow the programmer to express the operations he wants the computer to perform in a simple and concise fashion. However, often the programmer may not know precisely what operations he wants the computer to perform. It is here that these languages become inadequate, for they presuppose knowledge on the part of the human, and just facilitate transmission of this information to the computer.

Obviously when the user approaches the computer, he has some problem in mind, but it may be formulated only in terms of the results he wants achieved, and perhaps some of the goals along the way. His problem is thus not only of transmitting goals, but also one of defining more precisely the process to achieve these goals.

Newell[39] gives a spectrum of increasing specification as it goes on in the human, which we can roughly picture as follows:

goal → idea of solution → detail of solution → computer

At the far left, the human already has some way of recognizing the adequacy and desirability of results. Clearly several prior stages of ill-definition exist even further to the left. However, a long way also exists toward the right before the procedures for solving the problem are well enough defined to be communicated to a computer using current programming languages.

PILOT represents one approach to this problem. It leaves the language essentially unchanged; it is still a language of procedure, i.e., of detailed instruction. However, the human and the computer interact with very short delays, of the order of seconds. The language is highly incremental, so that the human can introduce new semantic as well as syntactic features, and it provides some way of talking about the changes and modifications one wishes to effect. Using PILOT, the human, still somewhat vague about just how he wants to proceed, operates experimentally. He constructs parts of programs that seem clearly needed, tries them out, organizes them into bigger routines, etc. In short, he finesses the restrictive effects of a language that demands explicit detail in favor of trial, rapid feedback, and correction.

However, this is not the only approach that can be taken to this problem. An alternative one would be to try to change the language, and move the communication boundary in the diagram above from the right side of the place marked "detail of solution to the left side. This approach is the "planning language approach" of Newell.[39] It attempts to understand the nature of communication between man and computer when he has only an idea of a solution. How can man and computers communicate before the man has worked out exactly what he wants to do? The solution: communication takes place in the language of plans. The man formulates only a general plan. The computer fills in the details and carries them out.

The situation is similar when we use high level languages for machine coding. The computer "fills in the details" of the program. However, while translating from ALGOL or FORTRAN

to machine language is algorithmic, to interpret a language of plans is, to some extent, to solve problems. That is, "the problem ... in developing a system that will take as input a linguistic expression for a plan is essentially one of artificial intelligence."[39] The real problems for the computer system are attaining all the unattained goals that comprise the plan. To do this the system must clearly be able to construct its own subgoals, and perhaps even be able to plan itself. This is far from what goes on inside of the FORTRAN compiler or LISP interpreter.

I feel that this approach complements the one taken by PILOT, and should certainly be explored. Any facility included in PILOT for interpreting plans would greatly aid the user. Since developing a "planning language" seems to be an artificial intelligence problem, perhaps the current PILOT system would be helpful for this purpose. In this way, we would be using PILOT to refine and improve itself.

Improving the Theory

The discussion of programming from the standpoint of block diagrams presented in Chapter 3 gives little more than a framework for introducing the concepts essential to PILOT. Much work remains to be done on defining more precisely what is meant by a procedure, and similarly, in what ways does one modify procedures. For example, we might start by attempting to formalize the block diagram by talking about its primitive elements and the allowable combinators.

Advances in this area would result in an immediate improvement to PILOT and similar systems. However, perhaps of even

greater significance, is the influence such work would have on the design and development of future programming languages. If we could obtain a really good formalization of the ideas discussed in this thesis, then it would be possible to construct languages and systems which would drastically simplify the task of programming. And until such time as these ideas are formalized, systems such as PILOT will only be a potpourri of ad hoc, although useful, subroutines.

Concluding Remarks

This thesis has described an approach to the solution of hard problems by computers. Basically, this approach, actually a philosophy, is: let the computer do it. Let the computer do anything and everything for you that is possible. The extra effort involved in automating even difficult processes will be returned in the freedom you receive to concern yourself with the problem.

PILOT is merely an example of this approach. If we were to implement a similar system on another machine, in another programming language, the resemblance to PILOT probably would be only superficial, although the concepts of procedures, essential variables, and advising might still be useful. However, the significance of PILOT is that it demonstrates the feasability and desirability of this approach. It clearly shows that it is possible to get computers to participate in, and cooperate with, research efforts in programming to a much greater extent than is now being done. I think we are far from developing a programming system that can truly be called symbiotic. However, PILOT is a step in the right direction.

This empty page was substituted for a blank page in the original document.

APPENDIX 1

SYMBOLIC DIFFERENTIATION IN LISP

Suppose that s is an expression to be differentiated with respect to the variable v, where s is represented in Polish prefix form, e.g., "3xy + 3yz + 3xz" is represented as (PLUS (TIMES 3 X Y) (TIMES 3 Y Z) (TIMES 3 X Z)). The following function, DIFF, will differentiate s with respect to v.

```
diff[s;v] = [   atom[s] ⟶ [eq[s;v] ⟶ 1; TRUE ⟶ 0]
eq[f[s];PLUS] ⟶ cons[PLUS;maplist[r[s]; λ [[x];diff[f[x];v]]]];
eq[f[s];TIMES] ⟶ cons[PLUS;maplist[r[s]; λ [[x];cons[TIMES;
      cons[diff[f[x];v];delete[f[x];r[s]]]]]]]        ]   *
```

To make DIFF completely general, we must add a fourth clause:

```
TRUE ⟶ cons[PLUS;map2[sublis[pair[f∘gradient∘f[s];r[s]];
      fr∘gradient∘f[s];r[s]; λ [[x;y];list[TIMES;f[x];diff
      [f[y];v]]]]]
                *
```

This clause allows us to introduce new operations to DIFF by making their gradients available to it, via the function GRADIENT. The argument of GRADIENT is the name of an operation, e.g., SIN, POWER, ARCTAN, etc., and its value is the gradient of

---

* f[s] denotes the first element of s and r[s] the rest of s, in other words, the functions CAR and CDR; the value of delete [x;y] is the list y with the element x deleted; map2 is similar to maplist but operates on two lists in parallel; "∘" denotes function composition.

that operation. GRADIENT thus plays the role of a table of derivatives.

The form of each gradient is a pair of lists of equal length, the first list being a list of variables, and the second list the partial derivatives with respect to those variables. For example, if we represent X^Y by (POWER X Y), the gradient of POWER is ((X Y) ((TIMES Y (POWER X (PLUS Y -1))) (TIMES (LOG X) (POWER X Y)))). This says that the derivative of X^Y with respect to X is X^{Y-1}, with respect to Y, $X^Y \log X$. Similarly, the gradient of SIN would be ((X) ((COS X))), etc.

If we restrict PLUS and TIMES to be binary operations, i.e., represent 3xy + 3yz + 3xz as (PLUS (TIMES 3 X Y) (PLUS (TIMES 3 Y Z) (TIMES 3 X Z))), then the gradient of PLUS is ((X Y) (1 1)), and the gradient of TIMES is ((X Y) (Y X)). In this case, the definition of DIFF can be written simply as:

```
diff[s;v] =   [   [atom[s] ⟶ [eq[s;v] ⟶ 1; TRUE ⟶ 0]
TRUE ⟶ cons[PLUS;map2[sublis[pair[f∘gradient∘f[s];r[s]];
         f∘gradient∘f[s]; r[s]; λ [[x;y];list TIMES;f[x],
         diff[f[y];v]]]]]     ]
```

APPENDIX 2

USING PILOT

The PILOT system is a collection of useful functions centered around the concept of advising, and the function ADVISE. This function is the only one crucial to the operation of PILOT. All of the other functions merely make it easier for the user to perform modifications. In this sense, these functions are not essential to the operation of PILOT, although it is difficult to see how PILOT would be useful, much less symbiotic, if these functions, or similar ones, were not available. This is particularly true with the translation scheme displayed in Chapters 5 and 6. The interface between PILOT and the user may and should be tailored to meet his own needs and desires. However, since the configuration and conventions I have found to be useful may provide a convenient starting point, I shall describe them in detail here. I must re-emphasize that this configuration, and these particular conventions, were adopted by me because they seemed useful and intuitive to me. I make no attempt to justify them, but merely present them to be taken at their face value.

SYSTEM

Normally, when a person uses LISP, he directs his requests to the EVALQUOTE operation. Computations are specified by giving this operator a pair consisting of a function and its arguments. EVALQUOTE evaluates this pair, types its value, and then awaits the next request.

To talk to PILOT, the user gives EVALQUOTE the pair "SYSTEM ()." This calls SYSTEM, the top level function of PILOT, which is a function of no arguments. SYSTEM plays a role in PILOT similar to that of EVALQUOTE. It accepts pairs and evaluates them in much the same fashion as EVALQUOTE. In fact, if the user specifies a function name and its arguments, the behavior of the two systems, PILOT and LISP itself, is indistinguishable. The user therefore could do all of his work while inside SYSTEM, although provision is made for exiting by typing "ok." In this case, SYSTEM returns the value NIL, and the user is back talking to EVALQUOTE, or wherever SYSTEM was called from.

The reason for introducing the function SYSTEM, is that the action of SYSTEM can be modified by advice. In fact, the construction of SYSTEM is designed for easy modification. The procedures that read and evaluate the EVALQUOTE pair are separated into two subfunctions. SYSTEM reads the first member of the pair, and calls SYS1, which reads the second member of the pair. SYS1 calls SYS2, which then evaluates the pair. This construction makes it easy to "drive a wedge" between SYSTEM and SYS1, or SYS1 and SYS2, and radically change the operation of the system.

What I have done in the current PILOT system is to advise SYS1, which has as its input the first member of the pair, and normally reads the second member giving both to SYS2, that when its argument is nonatomic, instead of reading the second member of the pair and going to SYS2, it should instead call the function DO. Thus if the user types "CAR ((A))" (two inputs), SYSTEM will type "A," having gone through the normal flow in SYS1 and SYS2. But if the user types (TELL FOO IF X IS LESS THAN

Y THEN QUIT) (one input), this expression will be given as input to the function DO, and SYS2 will not be entered.

It is the task of DO to determine and perform the operation specified by this input. DO does this by calling the function TRANSLATE, which transforms the input list using a sequence of FLIP transformations.

If all goes well, the transformed list, the value of TRANSLATE, will consist of the name of a function and its arguments. In this case, the first atom will be a special symbol "$." DO then treats the second atom as the name of a function, and the rest of the list as arguments. If the first atom is not "$," DO prints out "I DONT UNDERSTAND," followed by the offending list, which may have been partially transformed.

TRANSLATE, the function which does the translation, is also conceptually very simple. It obtains a list of rules for the translation process from the property list of the atom TRANSLATE, under the property RULES. Thus these rules are not intrinsic in the system, in fact, initially there are none, and the user can add them readily. TRANSLATE then calls a FLIP function TRANSFORM, giving it the rules and the input list. It is this latter function which does most of the work in translating. TRANSLATE also calls a function PARSE.

The entire structure of SYSTEM and its satellite functions is shown in Figure 5. The remainder of the appendix is devoted to describing the action of TRANSLATE, and the TRANSLATE RULES, in greater detail.

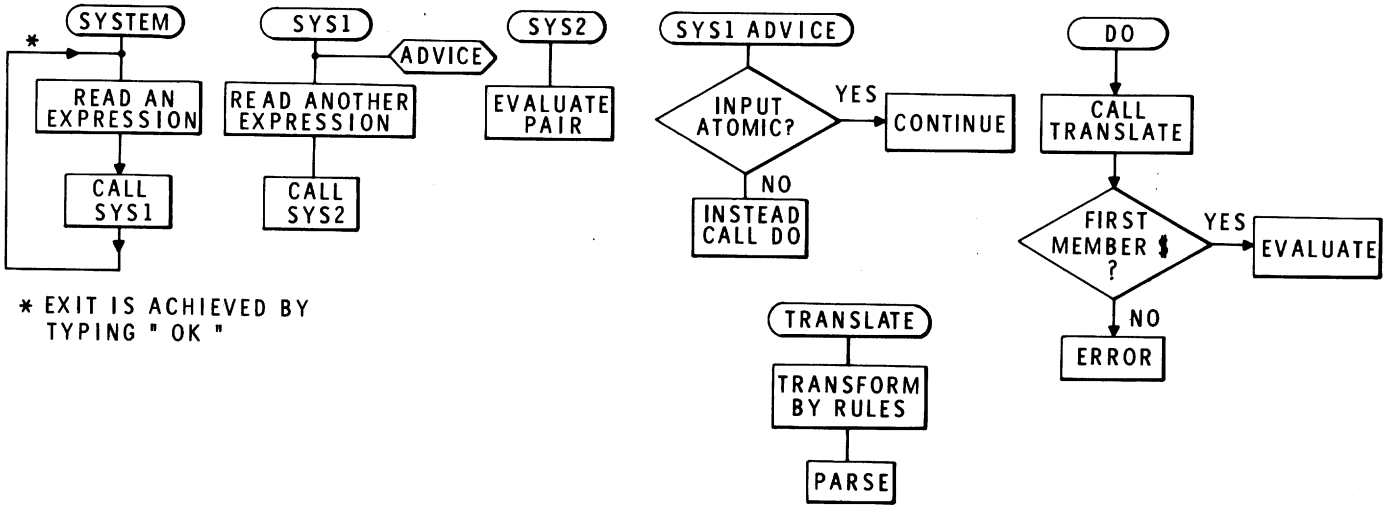


FIG. 5 FLOW CHART OF PILOT

TRANSLATE

TRANSLATE is designed to allow the user to specify an operation in what looks like a sentence, interspersed perhaps with some LISP expressions. The translation process operates by collapsing sections of this sentence into LISP computations until, if successful, all that is left is a single computation. In this case, the form of the list will be ($ function-name function arguments). For example, if the user wishes to modify the function FOO so that after it is evaluated, if its VALUE is not a member of the list x, or if it is greater than y, FOO should return with twice its VALUE, he might type (TELL FOO AFTER IF VALUE IS NOT A MEMBER OF X OR VALUE IS GREATER THAN Y THEN RETURN WITH (TIMES 2 VALUE)). This becomes (TELL FOO AFTER IF (NULL (MEMBER VALUE X)) OR VALUE IS GREATER THAN Y THEN RETURN WITH (TIMES 2 VALUE)), and then (TELL FOO AFTER IF (NULL (MEMBER VALUE X)) OR (GREATERP VALUE Y) THEN RETURN WITH (TIMES 2 VALUE)), and then (TELL FOO AFTER IF (OR (NULL (MEMBER VALUE X)) (GREATERP VALUE Y)) THEN RETURN WITH (TIMES 2 VALUE)), and then (TELL FOO AFTER IF (OR (NULL (MEMBER VALUE X)) (GREATERP VALUE Y)) THEN (LIST (TIMES 2 VALUE))), and then (TELL FOO AFTER (COND ((OR (NULL (MEMBER VALUE X)) (GREATERP VALUE Y)) (LIST (TIMES 2 VALUE))) (T NIL))), and finally ($ SYSTEM1 FOO AFTER (COND ((OR NULL (MEMBER VALUE X)) (GREATERP VALUE Y)) (LIST (TIMES 2 VALUE))) (T NIL))).

If the user had typed any of the intermediate expressions as input directly, the end result would have been the same. If he added a rule which transformed (... TWICE xxx ...) into (TIMES 2 xxx), he could have written (TELL FOO AFTER IF VALUE IS NOT A MEMBER OF X OR VALUE IS GREATER THAN Y THEN RETURN WITH TWICE VALUE).

There are two processes that take place inside of TRANSLATE. The major one, of course, is the transformation of the input list according to TRANSLATE RULES. However, since TRANSLATE is called from inside of these rules, at various levels, to perform translations on parts of the input list, a parsing feature has been added so that where the user wishes to express a LISP computation, because he has not included a translation rule that will handle it, he can do so with a minimum of parentheses. I shall describe this operation first because it is fairly simple.

PARSE is a function which utilizes information about the number of arguments of a function is order to insert parentheses in an otherwise unstructured list. For example, PARSE transforms (CONS CAR X CDR Y) into (CONS (CAR X) (CDR Y)). PARSE has no effect on lists which do not begin with function names, or for other reasons are not appropriate for parsing, e.g., they are already parsed. For those lists which do look like they should parse, but do not, PARSE gives appropriate errors. The application of parsing permits a vast reduction in the number of parentheses the user must employ, and greatly increases the readability (and writeability) of LISP expressions. Since PARSE is called from TRANSLATE, any expression which will normally be translated will also be parsed. Thus, in the previous example, the user could write (TELL FOO AFTER IF (OR NULL MEMBER VALUE X GREATERP X Y) THEN (LIST TIMES 2 VALUE)), and the end result would be the same.

TRANSLATE RULES

TRANSLATE makes use of a FLIP function TRANSFORM to transform the input list. The input to TRANSFORM is an item to be transformed and a list of rules. Each rule consists of three

parts, a pattern for matching, an (optional) format for constructing, and an (optional) label for transferring control. If the input matches the pattern, it is transformed according to the format, if any, and control goes to the labeled rule, or else to the next one. If the input does not match, control goes to the next rule regardless. TRANSFORM thus acts very much like METEOR, and allows one to write little COMIT-like programs using FLIP. Exit is achieved either by "dropping off" the end of the list of rules, or by going to a fictitious label EXIT. One can also return to the top of the list of rules by going to TOP. With my translation scheme, this is done after every successful match-construct operation, except those which produce the special "$" symbol indicating the transformation process is complete. For these, the transformation process is terminated by a call to EXIT.

The philosophy behind each rule has been that where there is no ambiguity, I should be allowed to suppress parentheses. For example, there is a rule which transforms (X IS NULL) into (NULL X), and a rule which transforms (IF xxx THEN yyy ELSE zzz) into (COND (xxx yyy) (T zzz)). Thus (IF (X IS NULL) THEN yyy ELSE zzz) becomes (COND ((NULL X) yyy) (T zzz)). However, I can also write (IF X IS NULL THEN yyy ELSE zzz), because the IF-THEN rule will not be applied until "X IS NULL" is changed to (NULL X).

I should mention that when the user writes (IF X IS NULL THEN yyy ELSE zzz), instead of (IF (X IS NULL) THEN yyy ELSE zzz), he is sacrificing computation time for ease of writing and reading. This is because the translator will try to transform (NULL X) in (IF (NULL X) THEN yyy ELSE zzz), while making the

IF-THEN transformation.* However, for those rules where exit is normally achieved, with the special symbol "$" at the head of the list, more meaningful errors will be found and communicated to the user if he does not use extra parentheses.

For example, I have a rule which transforms (TELL FOO xxx) into ($ SYSTEM1 FOO BEFORE xxx), a call to one of the advising functions. If I say (TELL FOO (IFF X IS NULL THEN QUIT)), this will become ($ SYSTEM1 FOO BEFORE (IFF X IS NULL THEN QUIT)). Although I intended this piece of advice to be transformed into a COND, it wasn't because it contained IFF, instead of IF. However, the advice (TELL FOO IFF X IS NULL THEN QUIT) would not be transformed into a list with a "$" at the front, and DO would tell me about the error at this point - instead of LISP telling me later that IFF was not a bona fide function. At that point, I would have to figure out where the error came from, why, and what should have happened, whereas with DO I would know immediately.

One further point: occasionally, you must use extra parentheses - to indicate precedence. For example, (A AND B OR C) is transformed into (OR (AND A B) C). To make the AND relationship be the primary one, one must say (A AND (B OR C)). Normally, this cannot be avoided, because you have to decide which will take precedence - AND or OR. However, in some cases, by being clever about the particular translation rule, you can rule out most of the cases where extra parentheses would be necessary. For example, I have a transformation which takes something of the form (DO xxx) into (PROG2 xxx NIL). This is for ADVICE which

---

* This could be avoided by having TRANSLATE recognize when an expression had been translated previously.

should be executed, i.e., the xxx, but not affect the flow of computation, hence the NIL. However, if you type (DO IF X IS NULL THEN ...), you would get ((PROG2 IF NIL) (NULL X) THEN ...), which is nonsense. But, if this rule required xxx to be non-atomic, it would not operate in this case until after the IF rule had operated. Thus, you obtain the correct result: (PROG2 (COND ((NULL X) ...).

Of course, with a more sophisticated parsing scheme, one could back up from incorrect transformations, and much of this would not be necessary. However, TRANSLATE is extremely ad hoc, and it is interesting that it can do as well as it does.

* * *

I shall now describe each individual rule. For reference, a complete list is contained in Appendix 3. In this discussion rules marked with "*" differ from the corresponding ones actually in the system as listed in Appendix 3, although in all cases both perform identical operations. The difference is usually a question of efficiency versus intelligibility. The rules in the system are more efficient, their counterparts here more understandable.

```
SYSTEM1     (SYS1 (COND
      ((ATOM X) NIL)
      (T (LIST (PRINT (DO X))))))     BEFORE)
```

This is the initial modification to SYS1 that causes it to call DO in the event its input is nonatomic. SYSTEM1, as described earlier, has three arguments: the name of the function to be modified, here SYS1, the expression that constitutes the advice, which is the COND, and the place where the advice is to be inserted - BEFORE.

```
ADD     (((TRANSLATE $1 AS (EITHER
        ($1 / (NOT ATOM))
        ($))    ) ($ ADD (2 (EITHER
        (1 TOP)
        (1))    ) TRANSLATE RULES) EXIT) TRANSLATE RULES)
```

This input adds the first translation rule to the property list of TRANSLATE under the property RULES. The rule is ((TRANSLATE $1 AS ... EXIT) and transforms an input such as (TRANSLATE xxx AS yyy) into ($ ADD (xxx yyy TOP) TRANSLATE RULES). If the user wishes to specify a label for transfer if the rule matches, he can say (TRANSLATE xxx as yyy FOO) which is transformed into ($ ADD (xxx yyy FOO) TRANSLATE RULES). This rule is a device to enable one to add other rules without calling ADD specifically. Note that if this rule matches, no further transformations occur, i.e., if one says (TRANSLATE xxx AS yyy), it becomes ($ ADD etc..) and an exit occurs, as specified by the label EXIT.

```
(TRANSLATE
   (TELL $1 (EITHER
      (FIRST)
      ($1 / (ATOM) FIRST)
      ($1 / (ATOM))
      --)    $1)

AS
   ($ (EITHER
      (SYSTEM3)
      (SYSTEM3)
      (SYSTEM1)
      (SYSTEM1))    2 (= TRANSLATE -1) (EITHER
      ((= NORMAL))
      (1)
      (1)
      ((= NORMAL)))    ))            *
```

This rule has been entered using the (TRANSLATE xxx AS yyy EXIT) format made possible by the previous rule. Basically, once the input has been reduced to the form (TELL xxx zzz), (TELL xxx yyy zzz), (TELL xxx FIRST zzz) or (TELL xxx yyy FIRST zzz), translation is complete and SYSTEM1 or SYSTEM3 can be called.

```
(TRANSLATE
   (-- END (BACKTO BEG) $1 $ END --)
AS
   (-- (= TRANSLATE -3) --))
```

While it would be possible to make TRANSLATE be completely recursive and tear apart every list structure looking for something recognizable, this seemed to be a slow and inefficient process. In particular it penalizes the user for material already translated, i.e., legitimate LISP expressions. A problem arises, however, when it is desirable to have something be translated that is inside of an expression that itself normally would not be translated. BEG and END are here introduced as pseudo-parentheses to finesse this situation. By using BEG and END in place of parentheses, you can write everything at the same level so that translation will occur. This particular rule locates the first END, and then backs up to the first BEG before it, so that one can nest BEG's and END's.

```
(TRANSLATE
   (-- DO $1 / (NOT ATOM) --)
AS
   (-- (PROG2 (= TRANSLATE 3) NIL) --))
```

Frequently one wishes to perform a LISP computation in advice without disrupting the normal flow into or out of the function in question. Since the ADVICE function will interpret a non-null value as a signal to bypass the function, this computation is embedded in the form (PROG2 xxx NIL), where xxx is the desired computation. PROG2 is a LISP function which evaluates both its inputs and returns the second one, here NIL. This rule transforms DO xxx into (PROG2 xxx' NIL), where xxx' is the translation of xxx. Note that xxx is restricted to be non-atomic (see previous discussion, page 161).

```
(TRANSLATE
   (-- BIND (EITHER
      ($1 / (ATOM))
      ($1))    TO $1 --)
AS
   (-- (ATTACH (CONS (EITHER
      ((- CONS QUOTE 1))
      ((= TRANSLATE 1)))     (= TRANSLATE 5))  (CDDR
            HISTORY)
) --))
```

This rule allows the user to create and bind a new variable to some value; the binding will hold until the current function is left. This is done via a call to ATTACH giving it the name of the variable and its value, and (CDDR HISTORY) which is the appropriate place to ATTACH it, i.e., just after the function's name. One can specify the variable name directly or as a result of a computation.

```
(TRANSLATE
   (-- SAVE $1 on $1 --)
AS
   (-- (SETQ 5 (CONS (= TRANSLATE 3) 5)) --))
```

This transforms (... SAVE X ON Y ...) into (... (SETQ Y (CONS X Y)) ...) with appropriate translations.

```
(TRANSLATE
   (-- POP $1 --)
AS
   (-- (SETQ 3 (CDR 3)) --))
```

The inverse of the above operation.

```
(TRANSLATE
   (-- IGNORE --)
AS
   (-- NIL --))
```

This rule allows the user to use IGNORE for NIL. IGNORE has intuitive meaning when used in the context of advice, e.g., (IF X IS NULL THEN IGNORE) means if x is null then go on with the rest of the computation.

```
(TRANSLATE
   (-- QUIT --)
AS
   (-- (LIST NIL) -))
```

Similarly for QUIT and (LIST NIL) -- do not enter this procedure but instead return with NIL.

```
(TRANSLATE
   (X IS $ (EITHER
      (Y MEANS)
      (MEANS))     $1)
AS
   ($ ADD2 IS PATTERNS (3 (EITHER
      ($1)
      --)     ) FORMATS (-1 (QUOTE (= TRANSLATE
      (/T 2))) (EITHER ((QUOTE (= TRANSLATE -1)))
      --)     )))                              *
```

This rule makes it possible to add definitions such as (X IS GREATER THAN Y MEANS GREATERP), (X IS A NUMBER MEANS NUMBERP), etc., so that (IF X IS GREATER THAN Y AND Z IS A NUMBER ...) becomes (IF (GREATERP X Y) AND (NUMBERP Z) ...). The pattern for each transformation is stored on the property list of the atom IS under the property PATTERNS. The format is stored under the property FORMATS. The actual transformation is handled by the rule below:

```
(TRANSLATE
   (-- $1 IS (EITHER
      (NOT)
      --)    (EITHER
      (= GET IS PATTERNS))       --)
AS
   (-- ((EITHER
      (NULL ((EITHER
         (/T -2)
         (= GET IS FORMATS))       ))
      ((EITHER
         (/T -2)
         (= GET IS FORMATS))       ))   ) --))      *
```

This rule handles the transformations of both (... xxx IS ...) and (... xxx IS NOT ...). It gets the appropriate patterns from IS PATTERNS, and transforms according to formats on IS FORMATS.

```
(TRANSLATE
   (-- TO (EITHER
      (($1 / (NOT FUNCTIONP) $))
      --)    (REPEAT 1 $1 AND) $1)
AS
   (-- (PROG (EITHER
         ((/T 3))
         (NIL))
            (REPEAT (QUOTE (= TRANSLATE 1)))
            (= TRANSLATE -1))   ))
```

(... TO xxx AND yyy AND zzz ...) becomes (... (PROG NIL xxx yyy zzz) ...) as a result of this rule. This is so the user can specify a number of operations in one piece of advice. If PROG variables are necessary, they can be inserted just after the TO. The list of PROG variables can be distinguished from a form because it does not begin with a function. Thus (TO (x y z) xxx AND yyy AND zzz) becomes (PROG (x y z) xxx' yyy' zzz').

```
(TRANSLATE
   ($ / / (NILL) (REPEAT IF $1) (EITHER
      (ELSE $1)
      --)     (EITHER
      (END --)
      --)   )
AS
   (-- (COND
      (REPEAT ((= TRANSLATE 2) (= TRANSLATE 4)))
      (EITHER
```

```
   ((T (= TRANSLATE 2)))
   ((T NIL)))    )   (EITHER
-1
(2)
--)   ))
```

This rule translates IF THEN statements into conditionals. The form of the statement must be IF $1 THEN $1 IF $1 THEN $1 etc terminated either by END, or by the end of the list. This is to help the user catch errors at translation time. Thus (IF POP X THEN QUIT) becomes (COND ((SETQ X (CDR X)) (LIST NIL)) (T NIL)), but (IF POPP X THEN QUIT) does not translate. Note however that both (IF (POP X) THEN QUIT), and (IF (PÓPP X) THEN QUIT) will satisfy the IF-THEN rule. At some later point, however, a LISP error will occur because of POPP.

This rule also allows the user to insert an optional ELSE clause at the end of the IF-THEN statement. If none appears, (T NIL) is used.

The appearance of the NILL in $ / / (NILL) causes the rule to fail if the first IF-THEN is not correct. This is to avoid partial transformations of IF-THEN clauses inside of a longer statement, i.e., IF X IS NULL THEN Y IF A THEN B ELSE D becoming (IF X IS NULL THEN Y (COND (A B) (T D)))

```
(TRANSLATE
   (-- IF $1)
AS
   (-- (SYSTEM4 (= TRANSLATE 3))))
```

Occasionally, if a computation is not NIL, you want to return with that computation. Essentially, you want to write (IF xxx THEN xxx). However, this will cause xxx to be evaluated twice. One finesses this by writing simply (IF xxx), which

results in a call to SYSTEM4 which performs the appropriate action.

```
(TRANSLATE
   (-- AND (BACK 2) (REPEAT $1 AND) $1 --)
AS
   (-- (AND (REPEAT (QUOTE (= TRANSLATE 1))) (= TRANSLATE
   -2
)) --))
```

This rule handles expressions such as xxx AND yyy AND zzz ... which become (AND xxx' yyy' zzz'). It locates the first AND and then backs up. There may be some confusion between this rule and the rule which handles TO xxx AND yyy ... However, one can always use BEG and END or parentheses.

```
(TRANSLATE
   (-- OR (BACK 2) (REPEAT $1 OR) $1 --)
AS
   (-- (OR (REPEAT (QUOTE (= TRANSLATE 1))) (= TRANSLATE
   -2)) --))
```

Similar to above for AND. Note that (A AND B OR C) becomes (OR (AND A B) C), because the AND rule is before the OR rule. To produce (AND A (OR B C)) one writes (A AND (B OR C)).

```
(X IS A MEMBER OF Y MEANS MEMBER)

(X IS A NUMBER MEANS NUMBERP)

(X IS (EITHER
   (AN ATOM)
   (ATOMIC))     MEANS ATOM)

(X IS GREATER THAN Y MEANS GREATERP)

(X IS LESS THAN Y MEANS LESSP)

(X IS EQUAL TO Y MEANS EQUAL)

(X IS NULL MEANS NULL)
```

IS RULES in the system.

```
(TRANSLATE
   (-- RETURN WITH $1 --)
AS
   (-- (LIST (= TRANSLATE -2)) --))
```

If one wishes to return with xxx from a function, the advice should actually yield (LIST xxx). This rule transforms (... RETURN WITH xxx ...) into (... (LIST xxx) ...). Thus QUIT is the same as RETURN WITH NIL.

```
(TRANSLATE
   (DEFINE $1 (EITHER
      ((FEXPR) ($SET FOO (QUOTE -1)))
      ($1 ($SET FOO (= LENGTH (= CAR -1))))
      (($SET FOO (QUOTE 0))))    AS --)
AS
   ($ DEFLIST (((= CAR (= PUT (= FOO) (= CAR 2)
   ARGS)) (LAMBDA
 (EITHER
        ((L A))
        (1)
        (NIL))    (= TRANSLATE -1)))   ) (EITHER
      (FEXPR)
      (EXPR)
      (EXPR))   ))
```

This rule is to allow the user to avail himself of the translation process in defining new functions; you can write (DEFINE FOO AS ....). If no arguments follow FOO, NIL is supplied. If (FEXPR) follows FOO, (L A) are used as arguments and DEFLIST is called with FEXPR as its second argument. Otherwise EXPR is used. The reference to PUT in the format puts the number of arguments in the function being defined onto its property list so that PARSE can be used even though the function is not yet defined, e.g., in (DEFINE MEMBER (X Y) AS IF X IS EQUAL TO (CAR Y) THEN T ELSE (MEMBER X CDR Y)), PARSE would know how many arguments MEMBER had.

```
(TRANSLATE
   (-- INCREMENT $1 --)
AS
   (-- (SETQ 3 (ADD1 3)) --))
```

Transforms (... INCREMENT xxx ...) into (... (SETQ xxx (ADD1 xxx)) ...).

```
(TRANSLATE
   (-- (EITHER
      (SEARCHF)
      (COUNTF)
      (SEARCHP)
      (LISTP)
      (COUNTP))     $1 $1 / (NOT ATOM) (EITHER
      ($2 $)
      --)   )
AS
   (-- (2 3 (= CONS QUOTE 4) HISTORY) -1))
```

SEARCHF, COUNTF, SEARCHP, LISTP, COUNTP are functions useful in problem solving. SEARCHP, LISTP, COUNTP all take a list, a predicate, and an ALIST, as inputs. SEARCHP searches for an item that satisfies the predicate. LISTP lists all items that satisfy the predicate. COUNTP counts the number of items that satisfy the predicate. SEARCHF and COUNTF are similar except they take FLIP patterns instead of predicates, and therefore you can express relations between elements in the list. The ALIST is used for evaluating free variables. Since in the most frequent use of these functions you specify only the list and the predicate or pattern, this rule will quote the predicate or pattern, and supply HISTORY as the ALIST. (LISTF, another function, is not handled by this rule because it requires an extra argument that the other functions do not take.) You can specify an ALIST yourself, in which case this rule will not match.

```
(TRANSLATE
   (-- BREAK $1 --)
AS
   (-- (BREAK1 NIL T (ADVICE) (CONS ' (COND
      ((EQ (CAADR HISTORY) ' VALUE) (CAADDR HISTORY))
      (T (CAADR HISTORY)))   ) (CONS TYTAB (= CONS QUOTE 3)
))))
```

This rule allows you to insert a BREAK inside of advice. This is done via a call to the function BREAK1 described earlier. BREAK1 prints as its message the name of the function, which it obtains from the history list, and the message corresponding to the $1.

```
ADD    (CHANGE TRANSLATE RULES)

(TRANSLATE
   (CHANGE $1 (EITHER
      ($1 / (ATOM))
      --)    (REPEAT ((REPEAT $ $1 / (NOT ATOM)) $)))
AS
   ($ EDIT 2 (EITHER
      (1)
      ((= NORMAL))) ((REPEAT ((REPEAT M (/C 1 1) 1
      (= TRANSLATE 2)) (/C 1 2))) STOP)) EXIT))))
```

This rule result allows you to call EDIT giving it a sequence of changes. You can include items to be translated in these changes, e.g., (CHANGE FOO (INSERT IF X IS A MEMBER OF Y THEN QUIT BEFORE SAVE X ON Z)). The request "ADD (CHANGE TRANSLATE RULES)" serves to label this rule so that other rules (below) can transfer to this label instead of to TOP or EXIT.

```
(TRANSLATE
   (TELL $1 (EITHER
      ($1 / (ATOM))
      --)    ((EITHER
      (BEFORE)
      (AFTER)
      (INSTEAD OF))     (EITHER
      ($1)
      ($ ADVICE))   ) (EITHER
      ($1 / (ATOM))
      ($1))   )
AS
   ($ EDIT 2 (EITHER
      (1)
      ((= NORMAL)))    ($SET FOO (== (EITHER
      (/T 4 2)
      (1)
      ((= TRANSLATE 1) (BACKTO ADVICE) UP1))   ))
      ($SET FIE
(== (EITHER
      -1
      (1)
      ((ADVICE (** (= TRANSLATE 1)))))   )) (((EITHER
      (/T 4 1)
      (INSERT (** (= FIE)) BEFORE (** (= FOO)))
      (INSERT (** (= FIE)) AFTER (** (= FOO)))
      (REPLACE (** (= FOO)) WITH (** (= FIE))))   ))
      STOP))                              *
```

The CHANGE rule is designed primarily for editing. When the user wants to insert advice at some point, other than the beginning or end, or to replace one piece of advice with another, he uses this rule so that he does not have to specify the entire editing sequence. If the user specifies (BEFORE SAVE X ON Y ADVICE), this becomes (BEFORE (SETQ Y (CONS X Y)) (BACKTO ADVICE) UP1) (each piece of advice is a list headed by the atom ADVICE). If the user writes just (BEFORE FOO), EDIT will look for the label FOO instead of for a piece of advice.

```
(TRANSLATE
   (USE (EITHER
      ($1 FOR $1 $1)
      ($1 FOR $1)
      ($1 $1 FOR $1 $1)
      ($1 $1 FOR $1)
      ($1 $1))    (EITHER
      (BUT $)
      --) )
AS
   (CHANGE (/T 2 1) (EITHER
      ((= NORMAL))
      ((= NORMAL))
      (2)
      (2)
      (2)))        (SETQ NAME ($* QUOTE (EITHER
      (3)
      (3)
      (4)
      (4)
      (1))      )) (SETQ VAL ($* QUOTE (EITHER
      (4)
      ((= NORMAL))
      (5)
      ((= NORMAL))
      ((= NORMAL)))   )) (EITHER
      -1
      (2)
      NIL)      ))
```

This rule facilitates shifting advice from atom to atom and property to property. The various options are included to allow the normal mode to be suppressed. The USE instruction may also have a sequence of changes following it as in (USE xxx yyy BUT (REPLACE ...) (INSERT ...)). This rule transforms the input into the format for CHANGE and then goes to that label. The SETQ NAME and SETQ VAL are instructions for EDIT telling it where to put the edited list after it is finished.

```
(TRANSLATE
   (--  (EITHER
        ((EITHER
           (MAPLIST)
           (MAP))   )
        ((EITHER
           (MAPCAR)
           (MAPC))   ))   $1 ($1 / (NOT EQ FUNCTION)
           --) --)
AS
```

```
        (-- ((EITHER
              (/T 2 1)
              (MAPLIST)
              (MAP))    (= TRANSLATE 3) (FUNCTION (LAMBDA (X)
              (EITHER
                 ((= TRANSLATE (/T 4)))
                 ((= SUBST (CAR X) X (= TRANSLATE (/T 4)))))  ))

    ) --))
```

Frequently, one would like to process a list and perform some operation on each member of the list. The function MAPLIST, for example, has two arguments, a list, and a function. It constructs a new list in which each element is the result of applying the function to the corresponding position in the old list, e.g., (MAPLIST X (FUNCTION (LAMBDA (Y) (ADD1 (CAR Y))))), increments each element in a list. This rule is designed to make it easier to call such functions. It supplies the FUNCTION and LAMBDA, and also translates the functional argument. It also allows the user to specify whether the function is to be applied to the remainder of the list, as in MAPLIST and MAP (MAP only differs from MAPLIST in that it does not construct a new list), or the next element in the remainder of the list, as in MAPCAR and MAPC. Thus MAPC FOO (PRINT CADR X) becomes MAP FOO (FUNCTION (LAMBDA (X) (PRINT (CADR (CAR X))))).

```
        (TRANSLATE
           (NAME $1 $ IN $1 (EITHER
              ($1)
              --)   )
        AS
           (CHANGE -2 -1 (FLIP
              ($ (= TRANSLATE 3) (BACKTO ADVICE) UP1 $)
              ((QUOTE 1) ((QUOTE =) NAME1 (QUOTE -2) 2) (QUOTE -1))

        )   ))
```

This rule allows you to locate a particular piece of advice and define it as a function, so that the advice itself may

subsequently be advised. This is done by calling EDIT to locate the advice and replacing it with a call to the new function, which is then defined. Thus (NAME FOO1 SAVE X ON Y IN FOO) becomes (CHANGE FOO (FLIP ($ (SETQ Y (CONS X Y)) (BACKTO ADVICE) UP1 $) (1 (= NAME1 -2 FOO1) -1))), and control goes to CHANGE label. When EDIT is called, NAME1 will define FOO1 as the old advice.

```
   (DEFINE NAME1 (X Y) AS CONS ' ADVICE DEFINE LIST LIST Y
LIST ' LAMBDA NIL CDR UNFLATTEN X)

DEFLIST    (((NAME1 (LAMBDA (X Y) (CONS (QUOTE ADVICE) (DEFINE
 (LIST (LIST Y (LIST (QUOTE LAMBDA) NIL (CDR (UNFLATTEN X))
)))))))   ) EXPR)
```

This is the definition of the function NAME1 used in the above rule. It defines a piece of advice as a function. The value of NAME1 is (ADVICE name), which will be substituted for the original piece of advice.

```
(TRANSLATE
   ((EITHER
      (BEFORE $1)
      (AFTER $1)
       ($1 $1)
       ($1))    : --)
AS
   (TELL (EITHER
      (2 1)
      (2 1)
      (1 2)
      (1 (= NORMAL)))    DO --))
```

This rule allows you to write (AFTER FOO : INCREMENT X), instead of (TELL FOO AFTER DO INCREMENT X).

```
(CHANGE SYS1 (REPLACE PRINT UP1 WITH (PROG (Y)
   IF (ERSETQ PRINT DO X)
   THEN (TERPRI)
   IF   (PROG2 PRINTRED ' (EDIT OR FORGET IT) SETQ Y IF
 (SETQ Y (RDFLX)) IS EQUAL TO ' EDIT THEN (EDIT NIL X NIL)
IF Y IS EQUAL TO ' PILOT THEN (PROG2 SYSTEM X) IF (TRANSFORM
 Y GET ' EDIT ' RULES) IS NOT EQUAL TO Y THEN (EDIT NIL X LIST
 Y ' STOP) ELSE Y)
   THEN  (SYS1 Y)
   ELSE  (PRINT ' OK))   ))
```

This operation modifies the original advice on SYS1, which told it to call DO. The intent is to cause the system to allow the user to correct errors detected inside of DO. If an error occurs, the value of ERSETQ will be NIL, and (EDIT OR FORGET IT) is printed. The user can then modify his input, without re-typing the entire string. The user may type EDIT, to utilize EDIT on the input. A single editing operation, can be typed and will be recognized as such because it will be transformed by EDIT RULES. This editing operation will then be performed. The user can also type PILOT, in which case the system is called recursively. This allows the user to make modifications, and return for another attempt at translating the input which caused the error. This feature of "remembering" the last input if an error occurs is extremely useful and was suggested by Professor Minsky during a session with PILOT. It is illustrated in the example below.

```
(x is negative means minusp)
 (IS RULES)

cset (print *T*)
 *T*

(define abs (n) as
if n is negative then complement of n,
else n)
  I DONT UNDERSTAND:
    (DEFINE ABS (N) AS IF (MINUSP N) THEN COMPLEMENT OF
    N ELSE N)
*** ERROR CALLED

 (EDIT OR FORGET IT)
```

```
pilot (translate (- complement of $1 -) as (- (minus 4)
-))

 TRANSLATION:    (ADD ((- COMPLEMENT OF $1) (- (MINUS 4)
  -) TOP) TRANSLATE RULES)
 (TRANSLATE RULES)

ok

 TRANSLATION:    (DEFLIST ((ABS (LAMBDA (N) (COND
          ((MINUSP N) (MINUS N))
          (T N ))))) EXPR)
 (ABS)

EDIT (SYS1 BEFORE ((REPLACE PRINT UP1 WITH (PROG (Y)
            (COND
               ((ERSETQ (PRINT (DO X))) (TERPRI))
               ((PROG2 (PRINTRED (QUOTE (EDIT OR FORGET
                 IT)
)) (SETQ Y (COND
                  ((EQUAL (SETQ Y (RDFLX)) (QUOTE EDIT))
                    (EDIT
 NIL X NIL))
      ((NULL (EQUAL (TRANSFORM Y (GET (QUOTE EDIT
) (QUOTE RULES))) Y)) (EDIT NIL X (LIST Y (QUOTE STOP))))
                  (T Y))   )) (SYS1 Y))
               (T (PRINT (QUOTE OK))))   )   ) STOP))
```

This is the translation of the CHANGE SYS1 modification above.

APPENDIX 3

LIST OF MODIFICATIONS

```
SYSTEM1    (SYS1 (COND
      ((ATOM X) NIL)
      (T (LIST (PRINT (DO X)))))    BEFORE)

ADD    (((TRANSLATE $1 AS (EITHER
      ($1 / (NOT ATOM ))
      ($1))  ) ($ ADD (2 (EITHER
      (1 TOP)
      (1))   ) TRANSLATE RULES) EXIT) TRANSLATE RULES)


(TRANSLATE
   (TELL $1 (EITHER
      (FIRST)
      ($1 / (ATOM) (EITHER
         (FIRST)
         --)   )
      --)    $1)
AS
   ($ (EITHER
      (SYSTEM3)
      ((EITHER
         (SYSTEM3)
         (SYSTEM1))   )
      (SYSTEM1))   2  (= TRANSLATE -1) (EITHER
      ((= NORMAL))
      (1)
      ((= NORMAL)))   ))


(TRANSLATE
   (-- END (BACKTO BEG) $1 $ END --)
AS
   (-- (= TRANSLATE -3) --))


(TRANSLATE
   (-- DO $1 / (NOT ATOM) --)
AS
   (-- (PROG2 (= TRANSLATE 3) NIL) --))


(TRANSLATE
   (-- BIND (EITHER
      ($1 / (ATOM))
      ($1))    TO $1 --)
```

```
AS
   (-- (ATTACH (CONS (EITHER
      ((= CONS QUOTE 1))
      ((= TRANSLATE 1)))     (= TRANSLATE 5)) (CDDR HISTORY)
) --))


(TRANSLATE
   (-- SAVE $1 ON $1 --)
AS
   (-- (SETQ 5 (CONS (= TRANSLATE 3) 5)) --))


(TRANSLATE
   (-- POP $1 --)
AS
   (-- (SETQ 3 (CDR 3)) --))


(TRANSLATE
   (-- IGNORE --)
AS
   (-- NIL --))


(TRANSLATE
   (-- QUIT --)
AS
   (-- (LIST NIL) --))


(TRANSLATE
   (X IS $ (EITHER
      (Y MEANS)
      (MEANS))    $1)
AS
   ($ ADD (3 (EITHER
      ($1 ($* $SET FOO ((/T -1) (QUOTE (= TRANSLATE (/T 2))
) (QUOTE (= TRANSLATE (/T -2 -2))))))
      (($* $SET FOO ((/T -1) (QUOTE (= TRANSLATE (/T 2)))))
))   ) IS RYKES))


(TRANSLATE
   (-- $1 IS (EITHER
      (NOT)
      --)    (EITHER
      (= (COPYTOP (GET (QUOTE IS) (QUOTE RULES)))))    --)
AS
   (-- ((EITHER
      --
      (= (LIST (QUOTE NULL) FOO))
      (= FOO)) ) --))
```

```
(TRANSLATE
   (-- TO (EITHER
      (($1 / (NOT FUNCTIONP) $))
      --)    (REPEAT 1 $1 AND) $1)
AS
   (-- (PROG (EITHER
         ((/T 3))
         (NIL))
            (REPEAT (QUOTE (= TRANSLATE 1)))
            (= TRANSLATE -1))   ))


(TRANSLATE
   ($ / / (NILL) (REPEAT IF $1 THEN $1) (EITHER
      (ELSE $1)
      --)    (EITHER
      (END --)
      --)   )
AS
   (-- (COND
      (REPEAT ((= TRANSLATE 2) (= TRANSLATE 4)))
      (EITHER
         ((T (= TRANSLATE 2)))
         ((T NIL)))   )    (EITHER
      -1
      (2)
      --)   ))


(TRANSLATE
   (-- IF $1)
AS
   (-- (SYSTEM4 (= TRANSLATE 3))))


(TRANSLATE
   (-- AND (BACK 2) (REPEAT $1 AND) $1 --)
AS
   (-- (AND (REPEAT (QUOTE (= TRANSLATE 1))) (= TRANSLATE -2
)) --))


(TRANSLATE
   (-- OR (BACK 2) (REPEAT $1 OR) $1 --)
AS
   (-- (OR (REPEAT (QUOTE (= TRANSLATE 1))) (= TRANSLATE -2
)) --))
```

```
   (X IS A MEMBER OF Y MEANS MEMBER)

   (X IS A NUMBER MEANS NUMBERP)

   (X IS (EITHER
      (AN ATOM)
      (ATOMIC))    MEANS ATOM)

   (X IS GREATER THAN Y MEANS GREATERP)

   (X IS LESS THAN Y MEANS LESSP)

   (X IS EQUAL TO Y MEANS EQUAL)

   (X IS NULL MEANS NULL)


(TRANSLATE
   (DEFINE $1 (EITHER
      ((FEXPR) ($SET FOO (QUOTE -1)))
      ($1 ($SET FOO (= LENGTH (= CAR -1))))
      (($SET FOO (QUOTE 0))))    AS --)
AS
   ($ DEFLIST (((= CAR (= PUT (= FOO) (= CAR 2) ARGS)) (LAMBDA
 (EITHER
         ((L A))
         (1)
         (NIL))    (= TRANSLATE -1)))   ) (EITHER
      (FEXPR)
      (EXPR)
      (EXPR))  ))


(TRANSLATE
   (-- RETURN WITH $1 --)
AS
   (-- (LIST (= TRANSLATE -2)) --))


(TRANSLATE
   (-- (EITHER
      (SEARCHF)
      (COUNTF)
      (SEARCHP)
      (COUNTP)
      (LISTP))     $1 $1 / (NOT ATOM) (EITHER
      ($2 $)
      --)   )
AS
   (-- (2 3 (= CONS QUOTE  4) HISTORY) -1))
```

```
(TRANSLATE
   (-- BREAK $1 --)
AS
   (-- (BREAK1 NIL T (ADVICE) (CONS ' (COND
      ((EQ (CAADR HISTORY) ' VALUE) (CAADDR HISTORY))
      (T (CAADR HISTORY)))    ) (CONS TYTAB (= CONS QUOTE 3)
))))


ADD    (CHANGE TRANSLATE RULES)


(TRANSLATE
   (CHANGE $1 (EITHER
      ($1 / (ATOM))
      --)    (REPEAT ((REPEAT $ $1 / (NOT ATOM)) $)))
AS
   ($ EDIT 2 (EITHER
      (1)
      ((= NORMAL)))  ((REPEAT ((REPEAT M (/C 1 1) 1 (= TRANSLATE
2)) (/C 1 2))) STOP)) EXIT)


(TRANSLATE
   (TELL $1 (EITHER
      ($1 / (ATOM))
      --)    ((EITHER
      (BEFORE)
      (AFTER)
      (INSTEAD OF))     (EITHER
      ($1)
      ($ ADVICE))    ) (EITHER
      ($1 / (ATOM))
      ($1))    )
AS
   ($ EDIT 2 (EITHER
      (1)
      ((= NORMAL)))    ($SET FOO (== (EITHER
      (/T 4 2)
      (1)
      ((= TRANSLATE 1) (BACKTO ADVICE) UP1))    ))  ($SET FIE
(== (EITHER
      -1
      (1)
      ((ADVICE (** (= TRANSLATE 1)))))    ))  (((EITHER
      (/T 4 1)
      (INSERT (** (= FIE)) BEFORE (** (= FOO)))
      (INSERT (** (= FIE)) AFTER (** (= FOO)))
      (REPLACE (** (= FOO)) WITH (** (= FIE))))    )) STOP))
```

```
(TRANSLATE
   (USE (EITHER
      ($1 FOR (EITHER
         ($1 $1)
         ($1))   )
      ($1 $1 (EITHER
         (FOR $1 $1)
         (FOR $1)
         --)   ))    (EITHER
      (BUT $)
      --)   )
AS
   (CHANGE (/T 2 1) (EITHER
      ((= NORMAL))
      (2))    (SETQ NAME ($* QUOTE (EITHER
      (/T 2 3)
      (-2)
      (-1)
      ((/T 2 1)))   )) (SETQ VAL ($* QUOTE (EITHER
      (/T 2 3)
      (-1)
      ((= NORMAL))
      ((= NORMAL)))   )) (EITHER
      -1
      (2)
      NIL)   ))


   (DEFINE NAME1 (X Y) AS CONS ' ADVICE DEFINE LIST LIST Y
LIST ' LAMBDA NIL CDR UNFLATTEN X)

DEFLIST    (((NAME1 (LAMBDA (X Y) (CONS (QUOTE ADVICE) (DEFINE
 (LIST (LIST Y (LIST (QUOTE LAMBDA) NIL (CDR (UNFLATTEN X))
)))))))   ) EXPR)



(TRANSLATE
   (NAME $1 $ IN $1 (EITHER
      ($1)
      --)   )
AS
   (CHANGE -2 -1 (FLIP
      ($ (= TRANSLATE 3) (BACKTO ADVICE) UP1 $)
      ((QUOTE 1) ((QUOTE =) NAME1 (QUOTE -2) 2) (QUOTE -1))
)  ))



(TRANSLATE
   (-- (EITHER
      ((EITHER
         (MAPLIST)
         (MAP))   )
      ((EITHER
         (MAPCAR)
         (MAPC))   ))    $1 ($1 / (NOT EQ FUNCTION) --) --)
AS
   (== ((EITHER
      (/T 2 1)
      (MAPLIST)
      (MAP))    (=TRANSLATE 3) (FUNCTION (LAMBDA (X) (EITHER
         ((= TRANSLATE (/T 4)))
         ((= SUBST (CAR X) X (= TRANSLATE (/T 4)))))  ))
) --))
```

```
(TRANSLATE
   (-- INCREMENT $1 --)
AS
   (-- (SETQ 3 (ADD1 3)) --))


(TRANSLATE
   ((EITHER
      (BEFORE $1)
      (AFTER $1)
      ($1 $1)
      ($1))    : --)
AS
   (TELL (EITHER
      (2 1)
      (2 1)
      (1 2)
      (1 (= NORMAL)))     DO --))


   (CHANGE SYS1 (REPLACE PRINT UP1 WITH (PROG (Y)
      IF (ERSETQ PRINT DO X)
      THEN (TERPRI)
      IF   (PROG2 PRINTRED ' (EDIT OR FORGET IT) SETQ Y IF
   (SETQ Y (RDFLX)) IS EQUAL TO ' EDIT THEN (EDIT NIL X NIL)
IF Y IS EQUAL TO ' PILOT THEN (PROG2 SYSTEM X) IF (TRANSFORM
 Y GET ' EDIT ' RULES) IS NOT EQUAL TO Y THEN (EDIT NIL X LIST
 Y ' STOP) ELSE Y)
      THEN (SYS1 Y)
      ELSE (PRINT ' OK))   ))

EDIT (SYS1 BEFORE ((REPLACE PRINT UP1 WITH (PROG (Y)
            (COND
               ((ERSETQ (PRINT (DO X))) (TERPRI))
               ((PROG2 (PRINTRED (QUOTE (EDIT OR FORGET IT)
)) (SETQ Y (COND
                  ((EQUAL (SETQ Y (RDFLX)) (QUOTE EDIT)) (EDIT
 NIL X NIL))
      ((NULL (EQUAL (TRANSFORM Y (GET (QUOTE EDIT
) (QUOTE RULES))) Y)) (EDIT NIL X (LIST Y (QUOTE STOP))))
                  (T Y))   )) (SYS1 Y))
               (T (PRINT (QUOTE OK))))  )    ) STOP))
```

This empty page was substituted for a blank page in the original document.

BIBLIOGRAPHY

[1] Berkeley, E.C., and Bobrow, D.G., (eds.) The Programming Language LISP: Its Operation and Applications, Information International, Inc., Cambridge, Massachusetts, 1964

[2] Black, F., "A Deductive Question Answering System," Ph.D. Thesis in Applied Mathematics, Harvard University, Cambridge, Massachusetts, June, 1964

[3] Bobrow, D.G., "A Question Answering System for High School Algebra Word Problems," Proc. FJCC, Spartan Press, Baltimore, Maryland, 1964

[4] Bobrow, D.G., "Natural Language Input for a Computer Problem Solving System," Ph.D. Thesis in Mathematics, M.I.T., Cambridge, Massachusetts, September, 1964

[5] Bobrow, D.G., "METEOR: A LISP Interpreter for String Transformations," in [1]

[6] Bobrow, D.G., "The Comit Feature in LISP II," M.I.T. Project MAC Memo M-219, Cambridge, Massachusetts, February 18, 1965

[7] Bobrow, D.G., and Teitelman, W., "Format-Directed List Processing in LISP," BBN Report #1366, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, April 1966

[8] Bobrow, D.G., Darley, D., Murphy, D., Solomon, C.J., and Teitelman, W., "The BBN LISP System," BBN Report #1346, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, February 1966

[9] Cohen, K., and Wegstein, J.A., "AXLE: An Axiomatic Language for String Transformations," Comm. ACM 8, 11, November, 1965

[10] Crisman, P.A., (ed.) The Compatible Time-Sharing System - A Programmer's Guide, Second Edition, M.I.T. Press, Cambridge, Massachusetts, 1965

[11] Daley, R.C., and Garman, C., "ED: A Context Editor for Card Image Files," M.I.T. Project MAC Memo M-195, Cambridge, Massachusetts, March 15, 1965

[12] Edwards, D.J., and Minsky, M.L., "Recent Improvements in DDT" M.I.T. Project MAC Memo M-60, Cambridge, Massachusetts, November 15, 1963

[13] Engleman, C., "MATHLAB: A Program for On-Line Machine Assistance for Symbolic Computations," Proc. FJCC, Spartan Press, Baltimore, Maryland, 1965

[14] Fano, R.M., "The MAC System: The Computer Utility Approach," IEEE Spectrum, January, 1965

[15] Farber, D.J., Griswood, R.E., and Polawsky, I.P., "SNOBOL, A String Manipulation Language," JACM II, 1, 1964

[16] Feigenbaum, E., and Feldman, J., (eds.) Computers and Thought, McGraw Hill, New York, 1963

[17] Geldard, Frank A., (ed.) Communication Processes, Nato Conference Series, Vol. 4, Pergammon Press, New York, 1965, (Proceedings of a Symposium held in Washington in 1963)

[18] Gray, P., The Encyclopedia of the Biological Sciences, pp 984-986, Reinhold Publishing, New York, 1961

[19] Guzman, A., and McIntosh, H.V., "CONVERT" to appear in Comm. ACM, August, 1966

[20] Johnson, T.E., "Sketchpad III: A Computer Program for Drawing in Three Dimensions," Proc SJCC, Spartan Press, Baltimore, Maryland, 1963

[21] Kaplow, R., Strong, S., and Brackett, J., "MAP: A System for On-Line Mathematical Analysis," M.I.T. Project MAC Report TR-24, Cambridge, Massachusetts, January, 1966

[22] Licklider, J.C.R., "Man-Computer Symbiosis," IRE Transactions on Human Factors in Electronics, March 1960

[23] Licklider, J.C.R., and Clark, W.E., "On-Line Man Computer Communication," Proc. SJCC, Spartan Press, Baltimore, Maryland, 1962

[24] Licklider, J.C.R., "Problems in Man-Computer Communication," in [17]

[25] Licklider, J.C.R. Introductory Remarks in [17]

[26] Lindgren, N., "Human Factors in Engineering, Part II - Advanced Man-Machine Systems and Concepts," IEEE Spectrum, April, 1966

[27] Martin, W.A., and Hart, T., "Syntax and Display of Mathematical Expressions," M.I.T. Project MAC Memo M-257, July 29, 1965

[28] Martin, W.A., "Time-Sharing LISP," M.I.T. Project MAC Memo M-153

[29] Martin, W.A., "A Symbolic Mathematical Laboratory," Ph.D. Thesis in Electrical Engineering, M.I.T., Project MAC, Cambridge, Massachusetts, (In preparation)

[30] Maurer, W.D., "Computer Experiments in Finite Algebra," M.I.T. Project MAC Memo M-246, Cambridge, Massachusetts, June 14, 1965

[31] McCarthy, J., "Programs with Common Sense," Proc. Symp. on Mech. of Thought Processes I, HMSO, London, 1959

[32] McCarthy, J., "Recursive Functions of Symbolic Expressions," Comm. ACM, April, 1960

[33] McCarthy, J., et al, LISP 1.5 Programmers Manual, M.I.T. Press, Cambridge, Massachusetts, 1963

[34] Miller, G.A., "Man-Computer Interaction," in [17]

[35] Minsky, M.L., "Steps Toward Artificial Intelligence," in [16]

[36] Minsky, M.L., "A Selected Descriptor-Indexed Bibliography to the Literature on Artificial Intelligence, in [16]

[37] Minsky, M.L., "MATHSCOPE - A Proposal for a Mathematical Manipulation-Display System," M.I.T. Project MAC Memo M-118, Cambridge, Massachusetts, November 18, 1963

[38] Newell, A., Shaw, J.C., and Simon, H.A., "Report on a General Problem Solving Program," Intern. Confer. Information Processing, UNESCO House, Paris, 1959

[39] Newell, A., "The Possibility of Planning Languages in Man-Computer Communication," in [17]

[40] Project MAC Progress Report II, July 1964-July 1965, M.I.T. Press, Cambridge, Massachusetts, 1965

[41] Reintjes, J.F., and Dertouzos, M.L., "Computer-Aided Design of Electronic Circuits," presented at WINCON Confer., Los Angeles, California, February 2-5, 1966

[42] Rudloe, H., "Tape Editor," Program Write-up BBN-101, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, January 3, 1962

[43] Samson, P., "Music Compiler," Program Write-up PDP-1, M.I.T. RLE Computation Center, Cambridge, Massachusetts, Circa 1962

[44] Samson, P., "TECO," M.I.T. Project MAC Memo M-250, Cambridge, Massachusetts, July 23, 1965

[45] Schwartz, J.I., Coffman, E.G., and Weissman, C., "A General Purpose Time-Sharing System," Proc. SJCC, Spartan Press, Baltimore, Maryland, 1964

[46] Sutherland, I.E., "SKETCHPAD: A Man-Machine Graphical Communication System," Proc. SJCC, Spartan Press, Baltimore, Maryland, 1963

[47] Teitelman, W., "Real-Time Recognition of Hand-Drawn Characters," Proc. FJCC, Spartan Press, Baltimore, Maryland, 1964

[48] Teitelman, W., "EDIT and BREAK Functions for LISP," M.I.T. Project MAC Memo M-264, Cambridge, Massachusetts, September 1, 1965

[49] Teitelman, W., "FLIP - A Format List Processor," M.I.T. Project MAC Memo M-263, Cambridge, Massachusetts, September 1, 1965

[50] Wantman, M.E., "CALCULAID: An On-Line System for Algebraic Computation and Analysis," M.I.T. Project MAC Report TR-20, Cambridge, Massachusetts, September, 1965

[51] Yngve, V., "COMIT Programmer's Reference Manual, M.I.T. Press, Cambridge, Massachusetts, 1961

BIOGRAPHICAL NOTE

Warren Teitelman was born in Philadelphia on February 21, 1941. He attended Miami Senior High school, Miami, Florida, and received a B.S. degree in Mathematics from the California Institute of Technology in 1962, and an S.M. degree in mathematics from the Massachusetts Institute of Technology in 1963.

Mr. Teitelman held several scholarships at Caltech from 1958 to 1962, was elected to Tau Beta Pi, and upon graduating with honor, was awarded both National Science Foundation and National Defense Education Act Fellowships. At MIT, he was an NSF fellow and a research assistant with Project MAC. He was elected to Sigma Xi in 1963, and received the General Electric Prize in 1964.

Mr. Teitelman has been interested in automatic computation and computer programming since 1959. He has been employed by the Synchrotron laboratory and the Computation Center at Caltech; the System Development Corporation of Santa Monica, California; Bolt, Beranek, and Newman, Inc., of Cambridge, Massachusetts; and Information International, Inc., also of Cambridge. He has accepted a position as Senior Scientist at Bolt, Beranek, and Newman, beginning June, 1966

His publications include:

New Methods for Real-Time Recognition of Hand-Drawn Characters, Master's Thesis, Massachusetts Institute of Technology, Department of Mathematics, June 1963

"Real-Time Recognition of Hand-Drawn Characters," Proceedings of the Fall Joint Computer Conference, Spartan Press, Baltimore, Maryland, 1964

"FLIP-A Format List Processor" MIT Project MAC Memo Mac-M-263, September, 1965

"EDIT and BREAK Functions for LISP" MIT Project MAC Memo Mac-M-264, September, 1965

"Format Directed List-Processing In LISP" with D.G. Bobrow, BBN Report #1366, Bolt Beranek and Newman Inc. Cambridge, Massachusetts, April, 1966

"The BBN LISP System" with D.G. Bobrow, et al, BBN Report #1346, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts, February, 1966

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-32

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: 200 (207-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:_____

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): _____

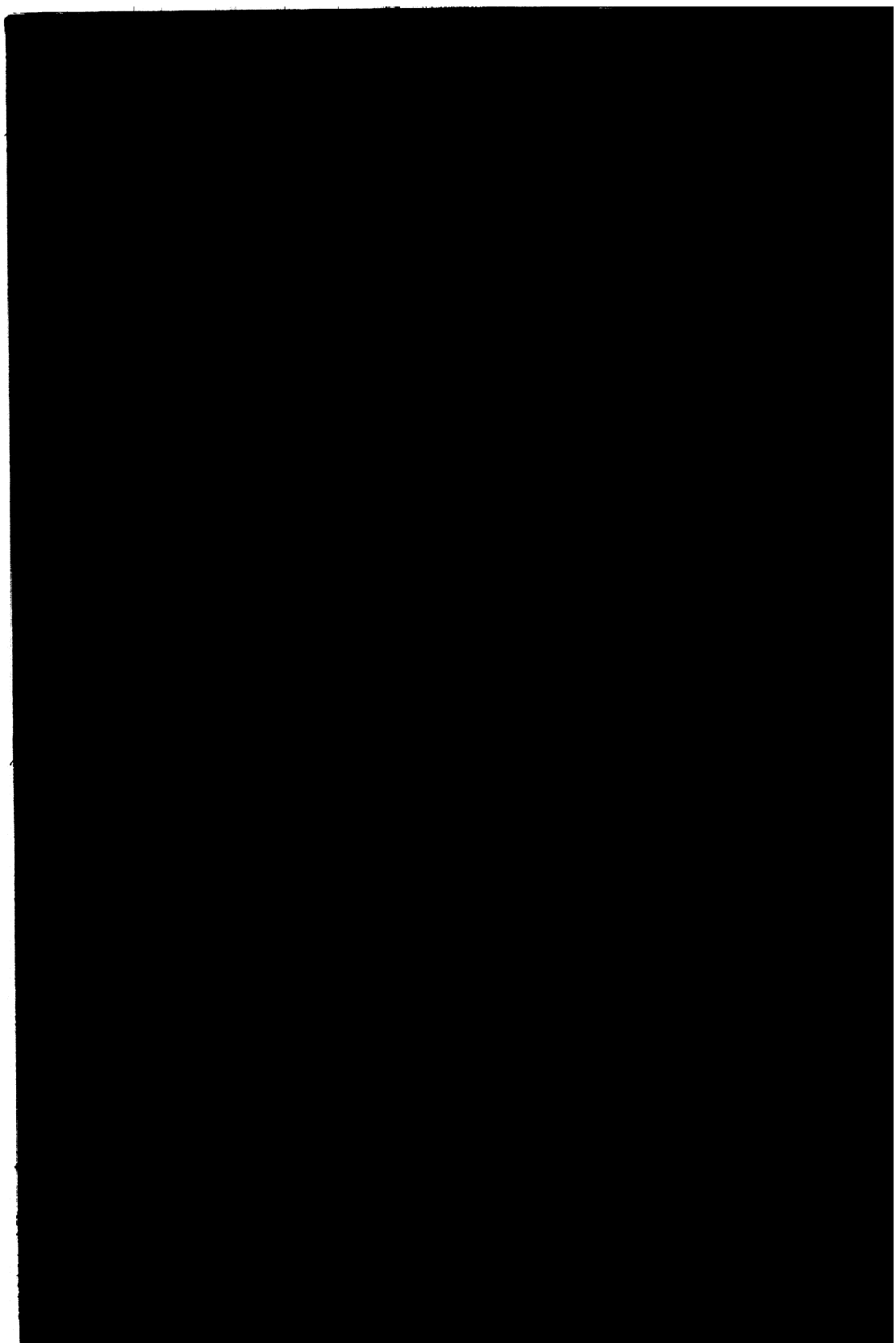
Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
IMAGEMAP: (1-200) UN# ED TITLE, BLANK PAGES, i-iv, 1-19, UN# BLK, 21-43, UN# BLK, 44-63, UN# BLK, 65-137, UN# BLK, 139-149, UN# BLK, 151-177, UN# BLK, 179-185, UN# BLK, 187-193, UN# BLK, (201-207) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGT'S (3)	

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 12/20/95 Date Returned: 12/28/95

Scanning Agent Signature: Michael W. Cook



Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/20/1995

M.I.T. Libraries
Document Services