TOWARD A PROGRAMMING LABORATORY

Warren Teitelman
Bolt Beranek and Newman Inc
50 Moulton Street
Cambridge, Massachusetts

Abstract

This paper discusses the feasibility and desirability of constructing a "programmlng laboratory" which would cooperate with the user in the development of his programs, freeing him to concentrate more fully on the conceptual difficulties of the problem he wishes to solve. Experience with similar systems in other fields indicates that such a system would significantly increase the programmer's productivity.

The PILOT system, implemented within the interactive BBN LISP system, is a step in the direction of a programming laboratory. PILOT operates as an interface between the user and his programs, monitoring both the requests of the user and the operation of his programs. For example, If PILOT detects an error during the execution of a program, it takes the appropriate corrective action based on previous Instructions from the user. Similarly, the user can give directions to PILOT about the operation of his programs, even while they are running, and PILOT will perform the work required. In addition, the user can easily modify PILOT by instructing it about its own operation, and thus develop his own language and conventions for interacting with PILOT.

Several examples are presented.

Introduction

The research described in this paper focuses on the programmer's environment. This term is meant to suggest not only the usual specifics of programming system and language but also such more elusive and subjective considerations as ease and level of interaction, "forgivefulness" of errors, human engineering, and system "Initiative." In normal usage, the word "environment" refers to the "aggregate of social and cultural conditions that influence the life of an individual." The programmer's enivronment influences, to a large extent determines, what sort of problems he can (and will want to) tackle, how far he can go, and how fast. If the environment is "cooperative" and "helpful" -- the anthropomorphism is deliberate — then the programmer can be more ambitious and oroductive. If not, he will spend most of his time and energy "fighting" the system, which at times seems bent on frustrating his best efforts.

One immediate goal to strive for is an environment comparable to that found in the well designed laboratory of the physical sciences. Such a laboratory usually contains equipment for many applications as well as facilities for designing and building new apparatus, or adapting that alreacfy present. In a large (well-funded) installation, the researcher will also often have available assistants for performing the "routine" tasks. For example, a chemist might simply request an analysis of a sample, and not have to itemize each step in the process. This type of assistant and assistance frees the researcher for problems more worthy of his attention.

Computer based systems have been constructed that create this type of laboratory environment for certain well-defined areas, e.g., mathematics [3,5,6], design of electronic circuits [7], and generalized graphical design [4,8], such as for aircraft, automobiles, bridges, etc.

These systems are organized to allow the computer to perform the routine work (where routine is a function of the sophistication of the system), while the user guides and directs the process at a relatively high level. For example, in the mathematical laooratory developed by Willian Martin, [5], the mathematician interacts with the computer by asking questions or making requests. The system employs graphical input and output (lipsntpen and display) to allow the mathematician to operate in an environment that closely resembles the pencil and paper with wnich he is already familiar. For output, the display utilizes subscripts, and observes the conventions concerning physical size, grouping, and placement of subexpressions which mathematicians have ado)ted to make It easier to read and comprehend mathematical formulae. For input, the mathematician can communicate directly with the computer via the light pen, either by writing new expressions, or by pointing to old ones, or portions thereof.

In a typical case, the user might be trying to find the solution of a differential equation. On the screen are displayed one or two equations, while the user has in his head the name of several other expressions or partial results already studied and filed away. The user decides to perform an action such as substituting a displayed equation, solving it for some variable, expanding some subexpression in a certain way, or perhaps asking to see something else. He makes the request using a combination of light-pen and key-board signals. These are encoded and transmitted to the system where the appropriate routines compute or retrieve the required new expressions and transmit them back to the display routines which then compile and display the desired new picture. In this way, the user can perform in a few minutes a long and involved analysis which, assuming he did not make any mistakes or lose track of what he was doing, might otherwise take him many hours.

This paper describes a step in the direction of such a laboratory for programming and programmers: the PILOT system. As with the mathematical laboratory, the goal is to allow the computer to perform the routine tasks while the user, in this case a programmer, is left free to concentrate on the more creative aspects of his problem, which is the writing and debugging of a program.

Most of the previous efforts bent at Improving the environment of the programmer have concentrated on providing and improving <u>packages</u>, such as editors, compilers, trace packages, display routines, etc. While a good deal of effort has been devoted to such facilities in the design of PILOT, the basic innovation of the PILOT system Is the emphasis placed on the <u>problem of making changes In programs</u>. The reason for this emphasis is that making changes in programs is the task that occupies most of a programmers time and effort, from the early stages In the development of programs when they consist primarily of correcting syntactical and simple logical errors in individual subroutines, to the final stages when the programmer makes the type of logical and organizational changes that affect many different parts of his program.

The problem of making changes to the PILOT system itself is handled as a special case of the problem of making changes in programs in general. Since PILOT is designed to facilitate making changes in programs, its tools and techniques can be applied directly to itself in what is essentially a bootstrapping process. The user can thus easily introduce new tools and/or modify existing ones to suit his own methods and problems, In short, tailor the performance of the system to suit himself. Puthermore, PILOT is designed with this in mind, so that it can cooperate with the user during this phase of the development.

<u>The PILOT System</u>

PILOT ic Implemented in the LISP programming language at Bolt Beranek and Newman Inc., Cambridge, Massachusetts. [1] Although there Is a PILOT subsystem in LISP, all of the features and tools described in this paper were incorporated directly into the BBN LISP system once their usefulness was established, and are now in general use by the entire community of LISP users. It is thus more meaningful to view PILOT as a conceptual system, a philosophy of design. It is this philosophy that we are trying to Impart, in the hopes that it may prove useful in the design and construction of systems in other languages. (*)

<u>Automatic Error Correcting</u>

The initial stages in the implementation of a large program are usually devoted to the writing and debugging of independent component routines. Only after these have been checked out, at least superficially, can the programmer begin to assemble the program and check for inter-routine problems. However, before the programmer can even begin to debug a routine, he must first get it to run, i.e., eliminate those syntactical and/or simple logical errors that cause complaints from the language or system in which he Is operating. Facilitating the correction of these lowest-level errors would improve the efficiency of debugging by allowing the programmer to proceed directly to higher level problems.

From the user's standpoint, clearly the best of all possible solutions would be for the system to correct these low-level errors automatically and continue with the computation. This is not far-fetched: a surprisingly large percentage of the errors made by LISP users are of the type that could be corrected by another LISP programmer without any information about the purpose or application of the LISP program or expression in

(*) LISP is especially suited for implementing a system such as PILOT because of the ease with which LISP programs can be treated as data by other programs. This capability is essential for creating tools which themselves will create and/or modify programs, an indispensible feature of a programming laborabory.

question,(*) e.g., misspellings, certain types of parenthesis errors, etc. If these corrections were performed automatically by a program that was called only when (after) an error occurred in the execution of a LISP program, it would in no way detract from the performance of the LISP system with debugged programs. Thus the efficiency of the error correcting program would not be a critical factor in its usefulness.

A primitive program which corrected certain types of spelling errors was implemented in PILOT and users were encouraged to experiment with it and comment on its features. As a result of this experience, we discovered that in order to be acceptable to users:

(1) The program must have a measure of how certain it is about the nature and correction of a mistake, and use this measure in determining the amount of interaction with the user.

(2) The program must be able to distinguish between significant and trivial corrections, and to be more cautious, i.e., more interactive, about correcting the former.

(3) The user must be able to specify to the program his degree of confidence in its ability to correct his mistakes, as reflected by the amount of interaction he desires.

(4) The user must be able to interrupt and/or abort any attempted correction.

(5) The user must be able to disable or overrule the entire correcting program if or whenever he wishes.

With these criteria in mind, a more sophisticated set of error correcting routines were implemented. These routines make up the DWIM package, for Do-What-I-Mean. The following output is representative of the kind of corrections and flavor of interaction of DWIM. User input is preceded by an arrow («-) .

In this example, the user first defines a function PACT of one argument, N, whose value is to be N factorial. The function contains several errors: TIMES and FACT have been misspelled. The 9 in N9 was intended to be a right parenthesis but the teletype shift key was not depressed. Similarly, the 8 in 8SUB1 was intended to be a left parenthesis. Finally, there are two left parenthesis in front of the T that begins the second clause in the conditional, instead of the required one.

(*) We conjecture that this is also true in other languages.

```
←DEFINE(((FACT (LAMBDA (N)
(COND ((ZEROP N9 1)
 ((T (TIMS N (FACTT 8SUB1 N]
(FACT)
←PRETTYPRNT((FACCT))
=PRETTYPRINT
=FACT

(FACT
  (LAMBDA (N)
    (COND
      ((ZEROP N9 1)
        ((T (TIMS N (FACTT 8SUB1 N))))))))
NIL
←FACT(3)
EDITING FACT ...
N9 >>--> N)
EDITING FACT ...
(COND -- ((T --))) >>--> (COND -- (T --))
TIMS=TIMES
FACTT=FACT
EDITING FACT ...
8SUB1 >>--> (SUB1
6

←PRETTYPRINT((FACT))

(FACT
  (LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (TIMES N (FACT (SUB1 N)))))))
NIL
```

After defining the function FACT, the user wishes to look at ls definition usinr PRETTYPRINT, which he unfortunately misspells. Since there is no function PRETTYPRNT in the system, an UNDEFINED FUNCTION error occurs, and the DWIM program is called. DWIM invokes its spelling cor- :ector, which searches for the best possible match a list of functions frequently used (by this user). Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRNT meant PRETTYPRINT, informs the user of this, and calls PRETTYPRINT.

At this point, PRETTYPRINT would normally print (FACCT NOT PRINTABLE) and exit, since FACCT has no definition. This is not a system error condition, but the DWIM facility is not restricted to just error conditions. DWIM modifies selected system functions, such as PRETTYPRINT and DEFINEQ, to make them cooperate more with the user. DEFINEQ is modified (by ADVISE, to be described later) to note any new functions defined by the user, and add them to the spelling list of user functions. Similarly, PRETTYPRINT is modified so that when given a function with no definition, it calls the spelling corrector. Thus, PRETTYPRINT determines that the user wants to see the definition of the function

PACT, not FACCT, and proceeds accordingly.

The user now calls his function FACT. During its execution, five errors are generated, and DWIM is called five times. At each point, the error is corrected, a comment made of the action taken, and the computation allowed to continue as if no error had occurred. Following the last correction, 6, the value of FACT(3), is printed. Finally, the user prints the new, now correct, definition of FACT.

In this particular example, the user was shown operating in a mode which gave the DWIM system the green light on all corrections. Had the user wished to interact more and approve or disapprove of the intended corrections at each stage, he could have operated in a different mode. Or, operating as shown above, he could have at any point aborted the correction or signalled his desire to see the results of a correction after it was made by typing a ? on the teletype.

We have found from our experience with DWIM that most users are quite willing to entrust the program with the correction of errors, although each different user may want to operate with a different "confidence factor," a parameter which indicates how sure DWIM must be before making a correction without approval. Above a certain user-established level, DWIM makes the correction and goes on. Below another level, DWIM types what it thinks is the problem, e.g., DOES PRTYPNT MEAN PRETTYPRINT ?, and waits for the user to respond. In the in-between area, DWIM types what it is about to do, pauses for about a second, and if the user does not respond, goes ahead and does it. The important thing to note is that since an error has occurred, the user would have to intervene in any event, so any attempt at correction is appreciated, even if wrong, as long as the correction does not cause more trouble than the original to correct♦ Since DWIM can recognize the difference between trivial corrections, such as misspellings, and serious corrections, such as those involving extensive editing, bad mistakes are usually avoided. When DWIM does make a mistake, the user merely aborts his computation and makes the correction _he would have to make anyway_.

Error Handling in General

Certain types of errors that occur in the BBN LISP system cannot be handled by the DWIM program, e.g., NON-NUMERIC ARG, an error generated by the arithmetic functions, ARG NOT ARRAY, from the primitive array functions, etc. These are data type errors.(*) Another class of errors not handled by DWIM are the 'panic" errors: BP FULL, a complaint from the compiler meaning it has run out of binary program space; NONXMEM, an attempt to reference non-existent memory, usually caused by treating an array pointer as a piece of list structure; PDL OVFLW meaning pushdown list overflow, which usually implies a looping program, etc. Both data type and panic errors are not fixable, but they are _helpable_.

In our system, whenever an error occurs, it causes a trap to a user-modifiable program. (It is through this program that DWIM works.) If DWIM has not been enabled, or if the user aborts an attempted DWIM correction, or if DWIM cannot fix the error that has occurred, the system goes into a "_break_" and allows the user to Interact with the system While maintaining the context of the program in which the error occurred. This allows the user to intervene to try to rectify the problem, or to salvage what he can of the computation. While in the break, the system accepts and evaluates inputs from the teletype. Since all of the power of the system Is available to him, the user can examine variables, change their values, define and evaluate new functions, and even edit functions he is currently in. If another error occurs in a computation requested while in the break, the system goes into a second, lower break, and so on. Thus it is rarely the case that the results of a lengthy computation are lost by the occurrence of an error near its end.

The following example illustrates this process (user input is preceded by ":" or "*"). The user is running a large account>-Ing system, one of "whose subroutines is to perform the alphabetization of a list of names. The first indication of the presence of a logical error in the system is the message ATTEMPT TO CLOBBER NIL, meaning the program is attempting to change the value of NIL. The system goes into a break (1), and the user tries to determine where the error occurred by performing a backtrace (2). He sees that he is in the function ALPHA, interrogates the value of some of ALPHA'S variables (3), and realizes that the problem arose when his alphabetization routine attempted to compare the last element In the list to the one following it, I.e. and end-check problem. While still in the break, he proceeds to edit the function ALPHA (4). DWIM corrects his spelling, and since ALPHA happens to be

(*) Sometimes these errors are in fact caused by misspellings, but it is impossible to tell in general.

compiled, the editor retrieves its defining symbolic expression from its property list, typing PFOP (5) to call this to the user's attention. Consulting his listing, the user instructs the editor to find the expression beginning with COND that contains RETURN, (6) which he then prettyprints (7). The expression he wants is the one before this one, so he backs up (8), and makes the appropriate correction (9).He then recompiles ALPHA (10).

```
ATTEMPT TO CLOBBER NIL
TEITELMAN
IN  RPLACA

(RPLACA  BROKEN)                                1
:BT                                             2
RPLACA
ALPHA
ACCOUNTS2
ACCOUNTS1
ACCOUNTS


(SOBROW  OUILLIAN  MURPHY  BELL  NIL)           3
:Y
(AIL)
:Z
TEITELMAN
:(EDITF  ALHPA)                                 4
=ALPHA
PROP                                            5
EDIT
*(COND  CONTAINING  RETURN)                     6
PP                                              7
  (COND
     (FLG  (GO  LP) )
     (T  (RETURN  X) ) )
*BACK  PP                                       8
   (NULL  (SETO  Y  (CDR  Y) ) )
*(EMBED  SETQ  IN  CUR)                         1
*PP
   (CDR  (SETQ  Y  (CUR  Y) ) )
*OK
(ALPHA)
:(COMPILE  (QUOTE  (ALPHA)))                    10
LISTING?
ST
(OUTPUT  FILE)
NONE
(ALPHA  COMPILING)
(ALPHA  REDEFINED)
(ALPHA)
:?=                                             11
U  =  NIL
V  =  TEITELMAN

:(SETU  u  Y)                                   12
(NIL)
SEVAL                                           13
RPLACA  EVALUATED
:VALUE                                          14
(TEITELMAN)
:X
(BOBROW  QUILLIAN  MURPHY  BELL  TEITELMAN)
:OK                                             15
RPLACA
```

Now the user wishes to proceed with the computation, and so must correct the immediate error situation in the function RPLACA. He interrogates the value of RPLACA'S arguments by typing ?= (11), and changes the first argument to the value of Y (12). He then evaluates RPLACA (13), checks its value (14), and releases the break by typing OK (15).

As illustrated above, when an error occurs a user invariably wants to "look back" and see what happened earlier in the computation to cause the error situation. In BBN LISP, all information regarding the state of the computation in progress is stored on the push-down list and is explicitly available to the user and to user programs. In fact, without this capability, DWIM could only be used to correct certain trivial errors. We believe that for any type of programming laboratory environment, it is absolutely essential that programs be able to examine the state of the world at any point in the computation. In terms of LISP, this implies being able to examine the sequence of functions that have been called, and looking at variable bindings. Since the same variable may be bound in nested function calls a number of times, expecially during a recursive computation, the program must be able to specifv which binding of a variable it is referencing and be able to change a particular binding if necessary. For example, had X and Y been the name of RPLACA's arguments, the user should still be able to interrogate the value of X and Y in ALPHA. Finally, the program must be able £b cause the computation to revert back to a specified place on the push-down list regardless of the number and type of intervening functions that have been called All of these capabilities are present In our system.

User Breaks

The capability of stopping a computation and maintaining its context while executing teletype inputs is also directly available to the user as an aid in debugging in a variety of forms. [2] In the simplest case, the user can request that selected functions be modified to cause breaks whenever they are called, or only when a certain condition is satisfied, e.g.. (BREAK ALPHA (GREATERP (LENGTH X) 10)) will cause the alphabetization routine to break whenever it is given a list of length greater than 10. At this point the user can intervene and examine variables, edit functions, etc. exactly as with the case when an error occurs and the system causes a break.

Another way of using the break feature Is to specify that a function be "broken" only when it Is called from some particular function. For example, the user would be reluctant to break on the function SETQ, since almost every function uses it. However, he could (BREAK (SETQ IN ALPHA)), which would only break on calls to SETQ from within ALPHA. This is performed by calling the editor to find and modify all calls to SETQ inside of the function ALPHA. Thus the performance of SETQ Is not affected or degraded when called from any other function.

The user can also request that breaks be inserted at specified points inside of a function. The editor is then called (in this case the function must be an interpreted one, i.e. have an S-expression definition) to find the appropriate point and insert the break. For example, the user could (BREAKIN ALPHA (BEFORE (COND CONTAINING RETURN))), which would cause a break Just before executing the indicated form. Alternatively, he can call for a break by using the function HELP.

Finally, the user can request a break at any time during a computation by simply depressing a special key on the teletype. The next time a function is called, usually within a few milliseconds, a break will occur, and again the user can Intervene and examine the state of the computation, etc. These capabilities are invaluable for localizing problems in complex programs, especially recursive ones, and are powerful tools for finding where to make changes that complement those described below that provide how to make changes.

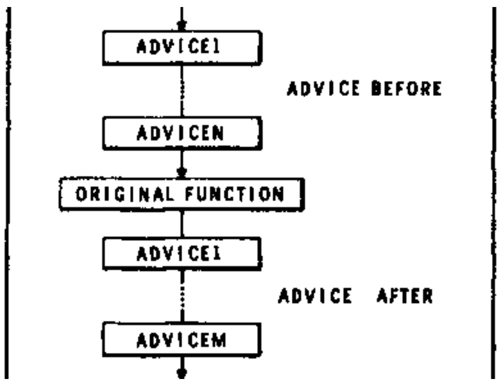
Advising

PILOT was originally motivated by the difficulties encountered In using computers for solving very hard problems, particularly those in the area of artificial intelligence.[9] These problems can be characterized as being 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. Even after he gets his program debugged, he continues to make alterations to see their effects. Only by experimenting with his working program can he evaluate its performance or hope to extend its generality. Since he cannot accurately predict the effect of changes on the behavior of the program because of its size and complexity, he must adopt the more pragmatic policy of "let's try it and see what happens." In short, he must be able to treat the computer as his laboratory.

Unfortunately, making changes in programs, especially large and complex programs, is often not a simple matter. Since they may require so much effort, many experimental changes are simply not implemented, with the result that the programs soon become "frozen." For this reason, considerable attention and effort in the design and development of PILOT has been devoted to the problem of making changes. One of the results is the concept of advising.

The operation of advising consists or modifying the interface between individual functions in a program, as opposea to modifying the functions themselves, which is called editing. The advantage of advising is that it allows the user to treat sections of his own (or someone else's) program as "black boxes," and to make modifications to them without concern for their contents. Since each modification is itself a small program, and modifications can be inserted so as to operate either before or after the original function would be run, advising is a very general and powerful concept.

Advising is carried out in LISP by creating a new function definition in which the original function definition is embedded, and surrounded by the "pieces of advice." This procedure is completely general: the function being advised can be arbitrarily large or small, complex or simple, compiled or interpreted, a system function or one of the user's own.



The individual pieces of advice are each LISP expressions, and so they to are completely general. Thus a piece of advice may simply change the value of some variable, or, at the other extreme, request a lengthy computation including perhaps calling the entire advised function recursively. Advice can also be given so as to bypass the entire advised function.

For example, the user could have repaired the problem In ALPHA shown earlier by giving the appropriate advice to RPLACA instead of editing ALPHA. Since RPLACA is called from many functions, the user would probably want to advise RPLACA IN ALPHA:

```
ADVISE((RPLACA IN ALPHA) (COND ((NULL U)
          (SETQ U Y))))
```

As with break, this would only modify the call to RPLACA from within ALPHA.

This operation demonstrates the advantage of advising. It allows the user to make online modifications quickly and simply. In addition to using it for correcting bugs, the user can perform modifications for the sake of experimentation, undo the modifications if he wishes, try out other configurations, etc., all without disruption to his high-level, problem oriented train of thought. Such disruption usually follows when implementing changes requiring a lengthy sequence of operations.

Note that advising complements rather than competes with editing as a way of making changes. In the early stages of debugging, the user is primarily attending to local phenomena in his program, and thus may find it natural to make changes by editing. In later stages, he considers his program more in terms of what each piece does, rather than how it does it, and here advising is the tool he wants to use for making changes.

<u>Advising as a Tool for Modifying the System</u>

Advising not only provides the user with a convenient tool for making changes in his own programs, but also with the- means for experimenting with and tailoring the system to his own particular tastes. For example, suppose a user wished to nodify PRETTYPRINT to print comments along the right had side of the page, where a comment was to be Indicated as an expression beginning with the atom *. Knowing that SUPERPRINT is the function that "does the work" of prettyprinting, he could

```
ADVISE(SUPERPRINT (COND ((EQ (CAR E)
        (QUOTE *)) (RETURN (COMMENT E)))))
```

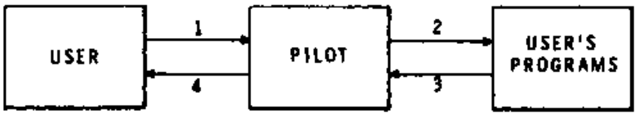
and then define the function COMMENT to do the appropriate formatting. (*)

TD The comment feature is now a part of our system. However, it was initially introduced in precisely this way, in order to evaluate its usefulness. Advising thus provides system designers with a quick means for trying out new features.

Admittedly this particular piece of advising requires the user to have some detailed knowledge of the workings of PRETTYPRINT. However, the important point is that by using ADVISE, changes <u>can</u> be easily effected, even with system functions where changes were <u>not</u> anticipated.

<u>Conversational Input</u>

PILOT" can be vTewed as an interface between the user and his programs. The following somewhat over simplified diagram illustrates the user-PILOT-program configuration:



Most of the effort in PILOT is concentrated at interface 2 and 3. However, in order to be really effective, a programming laboratory should not only provide the means whereby changes can be effected immediately, but also in a <u>way that seems natural to the user</u>. Accordingly, we have been experimenting with an English-to-LISP translating program that operates at interface 1, and translates the user's requests to PITOT into the appropriate LISP computation. The following dialogue gives the flavor of user-PILOT interactions obtained with this program. User input is preceded by ">."

```
←PILOT(T)
PROCEED:
>TELL PROGRESS: IF THE CANNIBALS OUTNUMBER
THE MISSIONARIES ON SIDE1, OR THE CANNIBALS
OUTNUMBER THE MISSIONARIES ON SIDE2,
THEN RETURN FALSE.

THE CANNIBALS OUTNUMBER THE MISSIONARIES
ON SIDE1 ??
>THE X OUTNUMBER THE Y ON Z MEANS
 Y IS CONTAINED IN Z AND THE NUMBER OF X
 IN Z IS GREATER THAN THE NUMBER OF Y IN Z.

THE NUMBER OF X IN Z ??
>DEFINE NUMBER, X Y, AS PROG (N)
   SET N TO 0;
LP: IF Y IS EMPTY THEN RETURN N,
    IF X IS EQUAL TO THE FIRST MEMBER OF Y
         THEN INCREMENT N;
    SET Y TO THE REST OF Y;
    GO TO LP.
(NUMBER)
>THE NUMBER OF X IN Z MEANS NUMBER X Z.
I UNDERSTAND.
>CONTINUE

I UNDERSTAND.
>CONTINUi-:

PROGRESS
```

The user instructs PILOT to advise the function PROGRESS with the statement beginning "TELL PROGRESS:.[11] PILOT recognizes this form of request, but does hot understand the part about outnumbering. The user then attempts to explain this with the input beginning THE X OUTNUMBER THE Y. This statement will cause an addition to PILOT'S already fairly extensive capability for converting English statements to LISP, so that PILOT will be able to understand expressions of this type encountered in the future. However, PILOT cannot interpret the phrase THE NUMBER OP X IN Z in this explanation, and so interrogates the user at this lower level. At this point, the user defines a new function NUMBER, and then explains the troublesome phrase in terms of this function. PILOT responds that it "understands." The user then instructs PILOT to continue with what it was doing, namely translating the explanation of OUTNUMBER. When this is completed, the user instructs PILOT to continue with the original request, which PILOT now successfully completes.

The current English-to-LISP translator contains a large assortment of useful, if ad hoc, transformational rules written in FLIP, [10], a string processing language embedded in the BBN LISP system. The set of FLIP rules can be easily expanded or modified. For example, the dialogue shown above resulted in rules for transforming expressions of the form THE X OUTNUMBER THE Y ON Z and for THE NUMBER OF X IN Z being added to the translator

In addition to the FLIP portion of the translating program, there is a post-processor which allows intermingling of LISP expressions with the English, as well as a sort of pidgin-LISP which looks like LISP with the parentheses removed. The translator also contains specialize information for dealing with quantifiers and and-or clauses. For example, the following expressions will be translated correctly into the equivalent LISP forms.

```
NO MEMBER OF X IS ATOMIC AND NOT NULL

THE FIRST ELEMENT OF X IS GREATER THAN THE
     SECOND AND NOT LESS THAN THE THIRD

THE FIRST ELEMENT OF SOME MEMBER OF X IS
     A NUMBER THAT IS GREATER THAN THE
     SECOND ELEMENT
```

The translator also "remembers" certain contextual information such as what was the last operation requested, what function it referred to etc. For example:

```
>TELL FOO: IF ITS FIRST ARGUMENT IS ATOMIC
     THEN RETURN IT,
FOO
>WHAT IS ITS SECOND ARGUMENT?
Y
```

We are not asserting that English is a good or even desirable programming language. However, if the user is thinking about his programs in English, then providing him the facility for expressing requests in English will allow him to concentrate more fully on the problem at hand.

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." [9] This statement was written in my Ph.D. thesis three years ago, and in the elapsed time, many of the goals established for improvements and additions to PILOT have been realized in our present system. But the statement is still true, and the process still continues.

One area of current interest is that of program-writing programs. Programming languages are.currently 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 many not _know_ precisely what operation he wants the computer to perform, although he may have a clear idea of what he wants the program to accomplish. That is, he may be able to give a description of its output, or the changes it should make in a data structure. This is not to say that the programmer could not construct the program. However, a system which could accept more goal-oriented descriptions of tasks and produce programs to accomplish them, even if only effective for simple, subroutine-level tasks, would further free its users for high-level operations. Such a system would require a fair degree of problem solving capability, and should have a sufficiently rich store of information about programming and programs to enable it to determine similarities in tasks. It should be able to adapt previously written or constructed programs to a new task. In other words, we are trying to construct a system that can handle more of the routine aspects of programming, in order to free the human to concern himself more with the creative aspects of the problem. This is the basic philosophy of the PILOT system: _let the computer do it_. The significance of PILOT is that it demonstrates this feasability and desirability of ths approach. Even in its current form, PILOT 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.

References

1. Bobrow, D.G., Murphy, D.L. and Teltelman, W. The BBN LISP System, April 1969.

2. Bobrow, D.G. and Teltelman, W. Debugging In an On-Line Interactive LISP, November 1967. Bolt Beranek and Newman Inc

3. Engelman, C. "MATHLAB 68" IFIP Congress 68, pp. B91-B95.

4. Johnson, T.E. "Sketchpad III: A Computer Program for Drawing In Three Dimensions," Proc. SJCC, Spartan Press, Baltimore, Maryland. 1963

5. Martin, W.A. Symbolic Mathematical Laboratory, Doctoral Dissertation, MIT, Cambridge, Massachusetts, January 1967. (also Report TM-36, Project MAC, MIT)

6. Maurer, W.D. "Computer Experiments in Finite Algebra," Comm. ACM, Vol. 9, No. 8, August 1966, pp. 589-603.

7. Reintjes, J.F. and Dertouzos, M.L. "Computer-Aided Design of Electronic Circuits," presented at WINCON Confer., Los Angeles, California, February 2-5, 1966.

8. Sutherland, I.E. "SKETCHPAD: A Man-Machine Graphical Communication System," Proc. SJ"CC, Spartan Press, Baltimore, Maryland. 1963

9. Teltelman, W. PILOT: A Step Toward Man-Computer Symbiosis, Doctoral Dissertation, MIT, Cambridge, Massachusetts, June 1966. (also Report TR-32, Project MAC, MIT)

10. Teltelman, W. Design and Implementation of FLIP, A LIsP Format Directed LisT Processor, BBN Report No. 1495, July