SESSION IX APPLICATIONS

RICHARD MILLWARD, *Brown University, Presider*

BANDAID: First steps toward an ICAI system

RICHARD B. MILLWARD
Brown University, Providence, Rhode Island 02912

BANDAID is an intelligent computer-assisted instruction (ICAI) system designed for students learning to program in BASIC. The major thesis is that beginning students need a friendly environment, that is, one in which information about the language and the system are immediately available. The system accepts most commands without switching levels: BASIC lines are parsed on input with very specific error messages. An editor allows corrections of BASIC lines. The program can be traced while running, and an indefinite number of HELP comments can be written for the system. Without leaving the BASIC environment, the user can interact with the CMS monitor of the main machine running the program. Although no English language comments or questions are allowed, the system is modular so that additions of restrictive dialogue interaction and guided tutoring are possible. Illustrations are presented showing the capabilities of the system. The paper also presents some of the implementation details of BANDAIID. A flexible system requires implementation in LISP, so examples of how the system programmer interacts with LISP during debugging are given. Although still incomplete, the system runs satisfactorily and can be used easily by beginning students and by psychologists interested in how people learn complex subjects.

Most of us remember the time when as youngsters, we set out to build our first "real thing." In my case, it was a wagon. Figure 1 presents a rough diagram of the result. It has most of the major components: four wheels, a steering device, a platform on which to sit, and a coat of paint. It did lack a brake, but since it never worked very well, that turned out not to be too serious a fault. Of course, in some ways it wasn't the final product that mattered, but rather the fun of building it. Every child learns from such projects, but what is learned probably differs greatly from child to child. Some learn to leave such engineering projects to those more competent. Others learn enough to start over, hop-

ing to do it right the next time, or at least better, since what is right is never really clear. Children learn that it is easier to imagine an ideal wagon than it is to make one. I think we would all agree that such projects are a worthwhile part of growing up.

There are psychological lessons to be learned from such a project. First, while there is no doubt that there is a goal for the project, the project is not specified exactly enough to be described in a top-down fashion. Therefore, the specificity of the goal must increase as the project progresses. Increased specificity is sometimes achieved by finding problems (bugs) in the implementation; eliminating the bugs leads to a more exact design. However, sometimes problems are so serious that they require a complete restructuring of the design. When programmers insist on programming in a top-down fashion, they are not referring to the creative aspects of

This project started in 1977 with a class of undergraduate students (Glen Kaufman, Scott Magoon, Louis Mazzucchelli, and Rebecca Moore) working on an artificial intelligence project. Although we were overly ambitious and the project barely got off the ground in one semester, the main structure of the system was defined. During the next year, only two of the students were actively involved and little progress was made. Last year, the IBM 360 at Brown was so overloaded that it was impossible to work with so large a system. However, Rebecca Moore took a few months last spring to put everything in shape, and her efforts were critical for the project. We are now at a turning point, since the new IBM 370 easily handles the large system, which is sufficiently free of bugs to make it a reasonable research tool. This research was supported in part by a grant from the National Science Foundation (BNS 75-08439A02) to the author.

Figure 1. A wagon built by a child.

programming. Creative programming often benefits from insight, but it generally requires some trial and error.

Second, complex objects are composed of parts. The ease of constructing the whole depends on understanding the parts and their interrelationships. Instead of trying to attach the wheels directly to the body of the wagon, one learns to make a separate wheel assembly that is attached later. Thus, there is a need both to analyze the whole object to find meaningful parts and to synthesize these parts into a whole. Building a wagon is not exclusively a top-down process, nor is it a matter of analysis or synthesis alone.

Before discussing the details of the BANDAID system, one further general remark should be made. A number of critics of artificial intelligence (AI) argue that building such "toys" is not scientific activity. At best, it is engineering that rests on principles of science. At worst, it is about as productive as the child's time spent on building wagons. But these critics seem to miss an important ingredient in all theoretical work: One way of viewing science is as a game of building toys. These toys are models of things we see in the world. A child imagines a fancy wagon and, by trying to build it himself, he learns where his image surpasses his knowledge. In an attempt to realize this model, the child learns to judge what makes a good model, where his own model falls short, and how to go about correcting it. Note that the child does not simply analyze his own store-bought wagon and set up hypotheses about its functioning or capability (Miller, 1978). So, we learn from constructing intelligent systems. In the field of AI, we have learned the inadequacies of most of our ideas about intelligent systems. We make small, seemingly insignificant steps in correcting the poor toys we have built and must often start over again and again. But each time we invent better tools, do the job a little better with more understanding of what the final structure ought to look like, and discover principles that help to guide future construction.

GOALS

BANDAID was designed to study how students without programming experience begin to learn about BASIC. Our intent was to develop a friendly, helpful, and natural environment in which to learn. For example, with no knowledge of assembly language, compilers, and interpreters, students often find it difficult to understand what they are doing when they edit a program, compile, and then run it. So, following the important insight of Kemeny in constructing BASIC, we wanted all aspects of the system to be at the same "level." This means that one edits, runs, and makes queries about the program without changing levels. That is, the monitor is transparent to the user.

We wanted to set the system up so that students would naturally hit upon questions and ways to formu-

late their needs that would be immediately acceptable to the system; in other words, we want to develop a programming system that required no manual. One of the more frustrating aspects of writing programs is not knowing how to do something and not knowing where to go for help, or even whether it can be done. For example, one may simply want to ask, "How do I express raising a number to a power?" Such questions required a natural language front end to the system, which we do not yet have. Our structure allows such a front end to be added eventually.

Finally, we wanted to make it easy for those developing the system to debug it. The design had to be flexible enough to allow new features to be added. When a system becomes complicated and large, these are necessary and not trivial requirements—They lead naturally to implementing the entire system in LISP. The BANDAID system was built as one large LISP program.¹ A discussion of some of the details of the program illustrates how the implementation provides for some of the above requirements.

PROGRAMMING IN BASIC WITH BANDAID

By calling BANDAID, the user is immediately put into an environment simulating DEC's OS/8 BASIC. Figure 2 presents a sample session in which a user types in a line number followed by a command and its arguments. A few points about these figures showing output from the program should be mentioned. First, following each equal sign on the terminal is a greater-than sign, which does not print on our output but, instead, causes a new line to be started. Thus, on the terminal, the line following the equal sign is on the same line as the equal sign. The capital-letter lines are those produced by the computer, and the (usually) lowercase lines are those typed by the user. The dollar sign, which appears all too frequently, represents 2 sec of IBM 370/158 CPU time. This is not a compiled system; if it were, it would be much less expensive. However, the system runs fast enough and is economical enough to be used as a research tool for studying how students learn to program. The semicolon indicates that everything to follow is a comment and is to be ignored by the program.

Newer BASIC interactive systems parse the lines as they are typed in. Thus errors are detected immediately, and no line is allowed that does not parse correctly. As Figure 2 illustrates, the usual RUN, SAVE, CLEAR, LOAD, and NAME are implemented. Variants such as RUNNH (no heading), STORE, SCRATCH, OLD/NEW, and RENAME are also accepted. A program is not stopped from running because it has no END statement since such a statement is a carryover from the days when programs were read in by cards and followed by data. The parsing program PARSE is discussed later.

The illustrated program in Figure 2 is simple; Figure 3 is more realistic. The LNORMAL program, which

```

; i am going to enter a simple program
=
rem this is a square root program
HUH?
=
10 rem This is a smple square root program
*** SYNTAX ERROR *** A LINE MUST START WITH A COMMAND
=
10 rem This is a simple square root program
=
20 print "input a number for square root"
*** SYNTAX ERROR *** A LINE MUST START WITH A COMMAND
=
20 print "input a number for square root"
=
30 print "input zero to stop"
*** SYNTAX ERROR ***
**END OF LINE IN STRING CONSTANT WITHOUT QUOTE**
=
30 print "input zero to stop"
=
40 input
*** SYNTAX ERROR *** MISSING VARIABLE
=
40 input a(i)
=
50 z = sqt(a)
*** SYNTAX ERROR *** SYNTAX ERROR IN EXPRESSION
=
50 z = sqr(a)
=
45 if a(i) = 0 then 100
=
60 print
=
60 print z
=
70 goto 40
=
100 stop
=
lis:
NONAME 11/05/79 02:24:07
10 REM THIS IS A SIMPLE SQUARE ROOT PROGRAM
20 PRINT "INPUT A NUMBER FOR SQUARE ROOT"
30 PRINT "INPUT ZERO TO STOP"
40 INPUT A(I)

45 IF A(I) = 0 THEN 100
50 Z = SQR(A)
60 PRINT Z
70 GOTO 40
100 STOP
READY
=
run
NONAME 11/05/79 02:24:27
WARNING - MISSING "END" STATEMENT ASSUMED
INPUT A NUMBER FOR SQUARE ROOT
INPUT ZERO TO STOP

4
1.0
)
)
READY
goto 50
50 Z = SQR(A)
=
c /a/a(i)/
50 Z = SQR(A(I))
=
save
FILENAME?
=
sqrst
READY
=
run
SQRST 11/05/79 02:25:40
WARNING - MISSING "END" STATEMENT ASSUMED
INPUT A NUMBER FOR SQUARE ROOT
INPUT ZERO TO STOP

4
.2E+01

16
.4E+01

64
.8E+01

1024
.3199996E+02

0
READY
=

```

Figure 2. Entering a BASIC program in BANDAID.

generates normal deviates according to an algorithm in Knuth, is not undertaken by a beginning student until after at least a few weeks of study in an introductory course. Figure 3 lists the program after it has been loaded from the disk. It is run for two loops as a test. The "CMS" command allows us to issue commands to the IBM monitor, CMS. It indicates that there are 31 users on the system, a lower than average load. The "TIME" command issues the time, and the "COST" command provides the current amount of money spent in this session. The test is for 10 loops through the program, which is just about the default limit on the number of lines allowed before BANDAID asks the user if he wishes to continue. (This limit can be changed.) The cost is 13 cents, and the time 46 sec. For a student testing this program, the output is acceptably fast, and the cost is not prohibitive for research. Of course, both time and cost are high compared with BASIC programs coded in assembly language.

A novice programmer would not produce a program like LNORMAL without a great deal of help. As noted, one of our goals was to produce an environment in which the student could simply ask for help and learn how to do things without going to a manual. Nothing discourages a student more than being stopped by the lack of a simple command to do what he wants to do. Ideally, the student would say a few words, or the system would anticipate the need in some way so that he could easily proceed with the problem. Our simple and crude solution to this problem is providing many "HELP" files that can be evoked by typing HELP X. Eventually, the system will accept a large number of responses as indicating a need by the student and print out the HELP files as appropriate. The critical factor is not to provide too much help, since that too frustrates a student—A 10-line message about something that is of no interest can be infuriating. Figures 4 and 5 present a few of the HELP files. The RUN files tell the student

```

load lnnormal
$
READY
=
list
LNORMAL 11/02/79 21:46:55
50 REM NORMAL DEVIATES BY POLAR METHOD.
60 REM PAGE 104 OF VOL 2 KNUTH.
70 PRINT "HOW MANY PAIRS OF NORMAL DEVIATES..."
80 INPUT N
100 DIM X(20),Y(20)
105 RANDOMIZE
110 FOR I = 1 TO N
120 V1 = 2*RND(0) - 1
130 V2 = 2*RND(0) - 1
140 S = V1**2 + V2**2
146 IF S >= 1 GOTO 120
150 X(I) = V1*SQR(-2*(LOG(S)/S))
160 Y(I) = V2*SQR(-2*(LOG(S)/S))
161 PRINT X(I);Y(I)
165 NEXT I
180 PRINT "DO YOU WANT TO CONTINUE?"
185 INPUT AS
190 IF AS = "YES" THEN 80
200 STOP
300 END
READY
=
runnh
HOW MANY PAIRS OF NORMAL DEVIATES DO YOU WANT?

2
.5576343 .1893665E+01
.2153778 -.1156765
DO YOU WANT TO CONTINUE?

no
READY
=

```

```

cms q users
031 USERS, 000 DIALLED
=
time
TIME IS 21:52:20 EST FRIDAY 11/02/79 79.306
CONNECT: 00:13:20 VIRTCPU= 000:12.37 TOTCPU= 000:18.01
=
cms cost
$1.40
=
run
LNORMAL 11/02/79 21:52:30
HOW MANY PAIRS OF NORMAL DEVIATES DO YOU WANT?

10
.1507808E+01 .1928781E+01
.1739566E+01 .1092213E+01
.1283266E+01 .1227396E+01
-.2543413E+01 -.1402811E+01
.2490598 .2446073
.2585309E-01 .1687671
.2491839 -.2003615
.8567468 -.8532468
.1265749E+01 -.5840038
.536557 .8311256
DO YOU WANT TO CONTINUE?

no
****STATEMENT COUNT EXCEEDED****
DO YOU WISH TO CONTINUE EXECUTION?
=
yes
READY
=
cms cost
$1.53
=
time
TIME IS 21:53:06 EST FRIDAY 11/02/79 79.306

```

Figure 3. Loading a large program and running it to test BANDAID's efficiency.

how to run BASIC programs, including the TRACE function, the LIMIT on lines executed, and the DEBUG aid for more serious programmers. (These are illustrated in more detail below.) The HELP HELP file gets the student going in the first place. HELP STRING provides examples of comments for programming with strings. There is no limit on the number of such files, although we do not have them implemented as flexibly as we could. Eventually, the whole set of related concepts will be put into a data base that can be reached by English-language questions or can be evoked when an error in programming is noticed. There is still much to do to build such a knowledge base, and we do not pretend to have satisfactory solutions to all of the questions. For example, we have not decided whether it would be better to organize the knowledge in some hierarchy or by a set of productions all at the same level.

One important concern has been to keep the student from worrying about level switching. The EDITOR is not treated as a separate program but as a set of commands to be issued whenever needed. Figure 5 shows the HELP EDITOR command and its use. The commands TOP, BOTTOM, PRINT N, GOTO N, UP N, DOWN N, FIND /X/, and CHANGE /X/Y/ are implemented and illustrated in their single-letter forms. The change command means that the user does not have to retype a whole line to make a single correction. Since BASIC lines determine their order, there is no real need for MOVE commands, and line deletion is accomplished by typing the line number.

The bottom of the HELP HELP file (Figure 4) suggests some of the powerful features of BANDAID

available to the systems programmer. He or she can enter LISP and work on the function definitions either by using the LISP editor directly or by adding debugging functions to already defined functions. Anything that can be done in LISP can be done without leaving BANDAID. This means that partial results, the state of various atoms, and so on, can be examined. A number of flags can be set so that errors, garbage-collection messages, and the like, are printed while the program is running.

Figure 6 presents another aspect of the system: We can call the CMS monitor easily. Two commands are of special note. WRITEALL allows us to trace the terminal output exactly as it occurs by writing every terminal line into a file. Figures 2-12 were created by WRITEALL. SUGGEST allows a user to make comments on the program. These comments are written into a file so that the systems people can read them later. We use it to make notes for later debugging. Other very useful CMS commands allow us to make a list of BASIC files in case the name of a program has been forgotten. Other CMS commands allow us to erase unwanted files, list files from the disk to see if they agree with the running version, obtain the time of day or the cost for the session, send messages to others, and execute almost any monitor command that does not evoke a program that interacts with the terminal. All of these are important because loading LISP and BANDAID systems takes time and costs money; it is more efficient to avoid exiting from BANDAID whenever possible.

For a beginner programmer, there is no function more important than tracing the program. Two kinds of

=
help lisp

LISP IS A LIST PROCESSING LANGUAGE DEVELOPED AT MIT,
FREQUENTLY USED IN ARTIFICIAL INTELLIGENCE.

=
help help

A BASIC STATEMENT MAY BE ENTERED SIMPLY BY TYPING IT IN,
E.G., 10 REM THIS PROGRAM PRINTS THE SQUARE ROOT OF 9
"LOAD <FILENAME>" READS A CMS FILE OF BASIC LINES
(THE COMMANDS "CLD" AND "NEW" MAY ALSO BE USED)
"SAVE" OR "SAVE <FILENAME>" SAVES THE CURRENT PROGRAM IN A CMS FILE
"LIST" OR "LISTNH" LISTS THE BASIC PROGRAM
"CLEAR" OR "SCRATCH" DELETES THE CURRENT BASIC PROGRAM
"NAME", "RENAME", AND "REPLACE" ARE ALSO IMPLEMENTED
"RUN" OR "RUNNH" RUNS YOUR BASIC PROGRAM
"FILE" PRINTS THE CURRENT FILE DATE AND TIME
"V" OR "VERSION" PRINTS THE DATE OF BANDAID'S MOST RECENT UPDATE
"CMS <COMMAND>" EXECUTES THE SPECIFIED CMS COMMAND
"CP <COMMAND>" EXECUTES THE SPECIFIED CP COMMAND
"TIME" PROVIDES THE CURRENT TIME
"SUGGEST" ALLOWS SUGGESTIONS TO BE MADE ABOUT BANDAID
"BYE" RETURNS YOU TO CMS

"EXPLAIN <KEYWORD>" PRINTS GENERAL INFORMATION ON A
NUMBER OF TOPICS--FOR A LIST, TYPE "EXPLAIN TOPICS"

FOR BANDAID SYSTEM PROGRAMMERS:

"/L" CALLS LISP
"/B" RETURNS TO BANDAID
"/T" RESETS FLAG FOR LISP ERROR HANDLING
"/X" TESTS BANDAID LISP ERROR HANDLING
"OUT" RETURNS TO TOP LEVEL OF LISP
"LISP <S1>" WILL EVALUATE THE S-EXPRESSION <S1>

help string

THERE IS ONE OPERATOR FOR STRINGS, &, WHICH INDICATES CONCATENATION.

IF A\$ = "ABC" AND B\$ = "DEF", THEN A\$&B\$ = "ABCDEF"

THERE ARE A NUMBER OF FUNCTIONS FOR STRING PROCESSING.

LEN(S\$)	= LENGTH OF THE STRING S\$
SFG\$(X\$,Y,Z)	= RETURN THE SUBSTRING OF X\$ STARTING AT POSITION Y AND ENDING AT POSITION Z
POS(X\$,Y\$,Z)	= STARTING AT POSITION Z, FIND THE POSITION AT WHICH Y\$ MATCHES A SUBSTRING OF X\$
DAT\$(X)	= THE DATE, MM/DD/YY
VAL(X\$)	= X\$ IS A STRING EXPRESSION FOR A NUMBER, VAL RETURNS THE NUMBER ITSELF
STR\$(X)	= X IS A NUMBER, STR\$ RETURNS THE STRING EXPRESSION FOR IT
ASC(X\$)	= X\$ IS A ONE-CHARACTER STRING, ASC RETURNS ITS DECIMAL EQUIVALENT AS REPRESENTED IN ASCII
CHR\$(X)	= X IS A NUMERIC EXPRESSION WHICH IS TREATED MODULO 63 AND THEN INTERPRETED AS A DECIMAL NUMBER FOR AN ASCII LETTER STRING.

FUNCTIONS WITH \$ RETURN STRINGS, FUNCTIONS WITHOUT \$ RETURN NUMBERS.
TO GET MORE INFORMATION ON THESE FUNCTIONS, CALL THE FUNCTION NAME
WITH HELP.

Figure 4. Three HELP files.

help run

RUN-TIME SYSTEM: The following options are available:

TRACE <ON,OFF>: Prints line number and line itself for all executed statements. Default is OFF.

LIMIT <number>: Specifies statement limit on execution. If exceeded, user is asked if execution should continue or abort. Default is 100.

DEBUG <ON,OFF> For debuggng purposes. Prints the CDR of the line, i.e. parms, command, and whatever other garbage we throw on. Default is OFF.

To use these, follow the RUN command with the option list, which is simply a series of options followed by their arguments. E.g.

RUN LIMIT 500 DEBUG ON

help editor

THE FOLLOWING EDITOR COMMANDS ARE SUPPORTED AND ARE

GENERALLY COMPATIBLE WITH THOSE OF THE CMS EDITOR.

THE STANDARD ONE-LETTER ABBREVIATIONS ARE RECOGNIZED.

CHANGE

UP

DOWN

NEXT

LOCATE

FIND

GOTO

TOP

BOTTOM

IN ADDITION, TYPING SIMPLY A STATEMENT NUMBER WILL DELETE

THAT STATEMENT FROM THE PROGRAM. TYPING "GETBACK <STATEMENT #>"

WILL RESTORE THE LATEST VERSION OF THE DELETED STATEMENT.

=

t

50 REM NORMAL DEVIATES BY POLAR METHOD.

=

b

300 END

=

t

50 REM NORMAL DEVIATES BY POLAR METHOD.

=

p 5

50 REM NORMAL DEVIATES BY POLAR METHOD.

60 REM PAGE 104 OF VOL 2 KNUTH.

70 PRINT "HOW MANY PAIRS OF NORMAL DEVIATES DO YOU WANT?"

80 INPUT N

100 DIM X(20),Y(20)

=

goto 150

150 X(I) = V1*SQR(-2*(LOG(S)/S))

=

u

146 IF S >= 1 GOTO 120

=

d 2

160 Y(I) = V2*SQR(-2*(LOG(S)/S))

=

c /)) /)) \ rem here is a second command on a line

160

Y(I) = V2*SQR(-2*(LOG(S)/S)) \ REM HERE IS A SECOND COMMAND ON A LINE

=

Figure 5. Helping the student edit and run his program.

```

-
;we have the facility of calling a number of monitor (cms) commands
=
;without leaving the BANDAID environment. Here are some examples.
=
time
TIME IS 21:41:48 EST FRIDAY 11/02/79 79.306
CONNECT= 00:02:48 VIRTCPU= 000:05.09 TOTCPU= 000:07.98
=
cms cost
$0.67
=
q v core
HUH?
=
;when i miss the cms command, it simply says HUH?
=
cms q v core
VADDR MISSING OR INVALID
=
cms q v
STORAGE = 00576K
=
cms list * basic
BNORMAL
DICK
ERATOSTH
LNORMAL
OCT31
PRIME
SQUARE
SQUARERT
STRING
TESTHILL
=
;that is a list of the BASIC program on my disk segment.
=
cms li * writeall
OCT30B
OCT30C
OCT31A
OCT31B
OCT31D
=
cms erase oct30b writeall
=
cms erase oct30c writeall
=
cms erase oct31* writeall
INVALID MODE 'WRITEALL'.
=
cms erase * writeall
=
cms li * writeall
FILE NOT FOUND
=
;what i cannot do is start up a different program.
suggest
ENTER SUGGESTIONS. TYPE "END" FOR LAST LINE.
?
there is a bug in goto. if the line does not exist, it goes to EOF
?
instead of saying there is no line of the next larger line. check it out.
?
end
THANKS FOR THE SUGGESTION.

```

Figure 6. CMS monitor commands: time, cost, core, list, erase, writeall, and suggest.

traces are allowed, but only the simple TRACE is of use to the naive programmer. This simply traces the program by typing the program statements line by line as the program runs. This version of TRACE is unsatisfactory because we should be able to turn it on selectively for some lines and not for others, and we should provide either the value of variables on each line (along with the result of the calculations of the line) or the values of selected variables when they change. Improvement of the present TRACE functions is not impossible, but the most efficient way to do it has not yet been determined. A command "SNAPSHOT V, X, A," which would print out the value of V, X, or A before and after it was changed, would be ideal.

Figure 7 illustrates the two traces we have implemented. The TRACE function provides the line-by-line output. the DEBUG function prints out the internal code of the line after it is parsed. DEBUG is useful for those who work with the system, since it indicates exactly what the INTERPRETER is working on.

Figure 8 illustrates the use of function definition and subroutines in the program FNORMAL. This program shows that the fact that we are simulating the BASIC interpreter does not limit the size or the commands that are allowed. Figure 9 presents an example of the string functions in operation. Although we have not added functions to those defined under DEC OS/8 BASIC, we could easily do so. Of particular interest is the addition of matrix operations.

Developing a system as complex as this is a challenge to one's debugging techniques. In developing BANDAID we felt it was important to trace the execution of LISP if we were to make any progress debugging the system. Although the LISP editor takes some practice to use efficiently, it is a powerful editor. Figure 10 presents an example of editing B-PARSE, the parsing routine for BASIC lines. We had the LISP code for B-PARSE in front of us so that we knew what we were trying to do. The LISP editor allows one to move from expression to expression until the right expression is located. Then changes can easily be made. Reading nested parentheses is difficult; the advantage of a LISP editor is that it moves by expressions, so that one always knows where one is. On the other hand, using the LISP editor requires learning another editor.

This LISP editor has an "UNDO" feature; if you make editing changes that you want to remove later, you can type "UNDO N" and the last N edit changes will be replaced in reverse order. Figure 10 has an example illustrating some of the features of the editor. The line (edit b-parse) calls the editor with the function B-PARSE. The entire function is displayed in a brief form, with "&s" replacing whole expressions. By typing "3 4 ?" the fourth subexpression of the third expression is printed on the terminal. (In this case it is the third & on the preceding line.) The "nx" command moves past the displayed expression to "GOODLINE." The "bk"

command moves back to the expression displayed two lines above, but without the "?" it is displayed in a brief form. Eventually, the line (FILE GOSUB GOTO REM) is found. The "i dim before file 1 : ?" command inserts dim before FILE. But it does not print. So, the insert command is repeated, but this time placing dim before GOSUB. That works. The next command, "0" backs up one expression and includes the old expression in the printed output. Notice that DIM was in fact inserted before FILE. So, "undo" is called and the DIM placed before GOSUB is removed. The command "ok" exits from the editor.

Another very attractive debugging facility allows adding a "BUG" to a function. A "BUG" is a function that is called before the "bugged" function is run (that is, on entry) and again after the function is run (on exit). The user can write any BUG function he likes and has access to the parameters being passed to the function and the value resulting from the function run. Figure 11 presents an example of a BUG on the function I-EVAL. The example shows that the comment "ENTRY-TO-I-EVAL" is printed, the arguments are printed, and a trace of a number of functions is initiated: SETQ, UNCONS, COND, CAR, APPLY, GET, DEFINEA. The example also illustrates one of the problems with such easy tracing facilities—They can be overdone. In this instance, the function was stopped before exiting. However, tracing common functions like SETQ or COND would be impossible unless they could be turned on as they are needed. Sometimes a function has to be traced in great detail because the source of the problem is simply not obvious. A BUG like this can be added to a function by a simple definition in LISP.

The above examples should provide some idea of what the system is designed to do. Although the system is by no means complete and is at present unsatisfactory in a number of ways, it is built in such a way that the needed new features can be added without too much trouble. It is not as if we had to recode everything to add one new feature. At the moment, we are developing a set of test programs that use all of the BASIC functions. When changes are introduced, these test programs can be run to assure that new routines have no unexpected side effects. Because the routines are well isolated, unpredicted side effects are unlikely, but they must still be checked for.

STRUCTURAL COMPONENTS OF BANDAID

BANDAID

The top-level program is BANDAID. It consists of a main loop that reads in a line from the terminal (READ LINE) and checks for a line number, special commands such as RUN, HELP, an edit command, a slash command, a CMS command, or BYE. Unless one of these is found, it returns HUH? Adding new features to this simple loop is simple. We expect to replace HUH?


```

-
run trace on debug on
LNORMAL 11/02/79 21:48:04
  50 REM NORMAL DEVIATES BY POLAR METHOD.
    (COMMANDS (REM) PARMS (NIL))
  60 REM PAGE 104 OF VOL 2 KNUTH.
    (COMMANDS (REM) PARMS (NIL))
  70 PRINT "HOW MANY PAIRS OF NORMAL DEVIATES DO YOU WANT?"
    (COMMANDS (PRINT) PARMS ((NIL (
HOW MANY PAIRS OF NORMAL DEVIATES DO YOU WANT?))))
HOW MANY PAIRS OF NORMAL DEVIATES DO YOU WANT?
  80 INPUT N
    (COMMANDS (INPUT) PARMS ((NIL (VAR 14 1 ,))))

1
  100 DIM X(20),Y(20)
    (COMMANDS (DIM) PARMS (((X 21) (Y 21))))
  105 RANDOMIZE
    (COMMANDS (RANDOMIZE) PARMS (NIL))
  110 FOR I = 1 TO N
    (COMMANDS (FOR) PARMS (((VAR 9 1 ,) (1) (VAR 14 1 , %) (1)
165)))
  120 V1 = 2*RND(0) - 1
    (COMMANDS (LET) PARMS ((VAR 22 3 , ! 2 RND 0 % * 1 - =)))
  130 V2 = 2*RND(0) - 1
    (COMMANDS (LET) PARMS ((VAR 22 4 , ! 2 RND 0 % * 1 - =)))
  140 S = V1**2 + V2**2
    (COMMANDS (LET) PARMS ((VAR 19 1 , ! VAR 22 3 , % 2 ** VAR 22
4 , % 2 ** + =)))
  146 IF S >= 1 GOTO 120
    (COMMANDS (IF) PARMS ((>= (VAR 19 1 , %) (1) 120)))
  150 X(I) = V1*SQR(-2*(LOG(S)/S))
    (COMMANDS (LET) PARMS ((X VAR 9 1 , % ! VAR 22 3 , % SQR -2
LOG VAR 19 1 , % % VAR 19 1 , % / * % * =)))
  160 Y(I) = V2*SQR(-2*(LOG(S)/S))
    (COMMANDS (LET) PARMS ((Y VAR 9 1 , % ! VAR 22 4 , % SQR -2
LOG VAR 19 1 , % % VAR 19 1 , % / * % * =)))
  161 PRINT X(I);Y(I)
    (COMMANDS (PRINT) PARMS ((NIL (X VAR 9 1 , % %) ; (Y VAR 9 1
% %))))
.6692631 .1948416E+01
  165 NEXT I
    (COMMANDS (NEXT) PARMS (((VAR 9 1 ,) 1 1 120)))
  180 PRINT "DO YOU WANT TO CONTINUE?"
    (COMMANDS (PRINT) PARMS ((NIL (DO YOU WANT TO CONTINUE?))))
DO YOU WANT TO CONTINUE?
  185 INPUT A$
    (COMMANDS (INPUT) PARMS ((NIL (VAR$ 1 1 ,))))

no
  190 IF A$ = "YES" THEN 80
    (COMMANDS (IF) PARMS ((= (VAR$ 1 1 , %) (YES) 80)))
  200 STOP

    (COMMANDS (STOP) PARMS (NIL))
READY

```

Figure 7. TRACE and DEBUG RUN options.

```

=
cms type fnormal basic
$

    10 REM NORMAL DEVIATES WITH FUNCTIN DEFINITIONS
    30 RANDOMIZE
    40 PRINT "HOW MANY DEVIATES?"
    45 INPUT N
    50 DEF FNV(X) = 2*RND(0) - 1
    60 DEF FNS(X,Y) = X**2 + Y**2
    70 DEF FND(X,Y) = X*SQR(-2*(LOG(Y)/Y))
    75 M = 0
    80 FOR I = 1 TO N
    90 V1 = FNV(X) \ V2 = FNV(X)
    100 S = FNS(V1,V2)
    110 IF S >= 1 GOTO 90
    120 X = FND(V1,S)
    130 Y = FND(V2,S)
    140 PRINT X;Y
    150 M = X + Y + M
    160 NEXT I
    170 PRINT "THE MEAN IS ="; M/N
    180 STOP
    300 END

=
runnh
    HOW MANY DEVIATES?

1
-.8413193 .2001041E+01
THE MEAN IS = .1159721E+01
READY

=
cms cost
$2.45

=
time
TIME IS 01:44:39 EST SATURDAY 11/03/79 79.307
CONNECT= 00:23:19 VIRTCPU= 000:24.82 TOTCPU= 000:35.43

=
runnh
    HOW MANY DEVIATES?

100
-.8169212 .1995928E+01
.....

.94908 -.1363634E+01
THE MEAN IS = -.2000003E-01
READY

=
cms cost
$4.16

=
01:45:56 MSG FROM GP500155: GOODNIGHT ALL! HAPPY GAME PLAYING.
time
TIME IS 01:46:42 EST SATURDAY 11/03/79 79.307
CONNECT= 00:25:22 VIRTCPU= 000:49.55 TOTCPU= 001:01.12
-

```

Figure 8. Using defined functions and timing a long run.

```

=
load string
$
  READY
=
list
  STRING ONE 02:33:47
  10 DIM A$(5,72), X$(72)
  15 DIM Y1$(72), Y2$(72)
  20 PRINT "INPUT 5 LINES CF TEXT"
  30 FOR I = 1 TO 5
  40 INPUT A$(I)

  50 NEXT I
  60 PRINT "ENTER A WORD TO DELETE FROM TEXT"
  70 INPUT X$
  80 L1 = LEN(X$)
  90 FOR I = 1 TO 5
  95 L2 = LEN(A$(I))
  100 Z = POS(A$(I),X$,1)
  110 IF Z <= 0 THEN 200
  120 Y1$ = SEG$(A$(I),1,Z-1)
  130 Y2$ = SEG$(A$(I),Z+L1,L2)
  140 A$(I) = Y1$ & Y2$
  150 GOTO 95
  200 PRINT A$(I)
  210 NEXT I
  220 STOP
  300 END
  READY
=
run
  STRING ONE 02:35:07
  INPUT 5 LINES CF TEXT

a string is a string is a string

i will string along with you

this line has no strings attached

here's a line tha+ will be unchanged

string
  ENTER A WORD TO DELETE FROM TEXT

string
  A IS A IS A
$
  I WILL ALONG WITH YOU
  THIS LINE HAS NO S ATTACHED
  HERE'S A LINE THAT WILL BE UNCHANGED

  READY

```

Figure 9. Using string functions.

with a function that will parse the input sentence so that a more natural dialogue will be allowed. But before this is accomplished, we need to know more about the kinds of questions and requests a naive user has. Then functions must be created for these requests.

PARSER

READ LINE returns the input line exploded (LINE) so that it can be checked character by character. The line number is padded with blanks on the left to right-justify the number, which is then quoted to make it an atom. The rest of the line after the line number is put on the P-LIST (thus called PLINE) of the line-number atom so that it can be printed out exactly as read in. All editing and parsing uses LINE to work on. The PARSER

begins by looking for a command. When it finds a command, it calls a function by that name (P-REM, for example), which processes the rest of the line according to the expectations of the command. A number of special functions were necessary for parsing. Since the input line is only a mixed list of single letters, numbers, and special characters, determining that a number variable or operator has occurred and that it is legal requires special routines. A number, for example, is put together as an ASCII string, which has to be converted to a number that LISP can work with. Expressions are also complex structures, but, because they can be converted to postfix notation and processed by a last-in-first-out (LIFO) stack, they are easy to deal with. Figure 12 presents our version of the expression evaluator.

Some aspects of the parsed line can be seen in Figure 7, where the PARMS component of the line is printed during the DEBUG trace. The REM line has no PARMS because the evaluation routine simply gets the comment from the original line stored on the P-LIST. Line 100 defines dimensioned variables X and Y. These become array variables in LISP: (X arg) returns the value of X(arg).

Line 150 shows a more complex function to evaluate. The PARMS line is ((X VAR 9 1 , % ! VAR 22 3 , % SQR -2 LOG VAR 19 1 , % % VAR 19 1 , % / * =)). This is processed from left to right as follows: X, VAR, 9, and 1 are pushed onto a stack. The comma is an operator that combines 9 and 1 into an index for the variable VAR. The % operator forms (VAR 9 1), which is EVALed by LISP into the value of I. (I is the 9th letter of the alphabet, and it has no digit following it. For example, I3 is a 9 5 , .) Now the stack is (X value-of-I). ! tells the system not to evaluate the two items on the stack at this time. It is paired with the = sign at the end. Moving left to right provides for the evaluation of V1 by combining VAR 22 3 , and %. Now the stack contains (X value-of-I value-of-V1). The SQR is pushed onto the stack, followed by -2, LOG, and the value of S. The stack is now (X value-of-I value-of-V1 SQR -2 LOG value-of-S), and the PARMS line is now (% VAR 19 1 , % / * % * =). The % operator combines the last two items on the stack into a LISP function call, that is, (LOG value-of-S), the value of which is then pushed on the stack. VAR 19 1 , % is then converted to value-of-S, and the last two items of the stack are (. . . value-of-LOG-S S). The next item on PARMS is the divide operator /, which causes division of the last two items. The remaining operators continue taking pairs of items off the stack and converting them to values until the = sign is reached. The stack is (X value-of-I ! value-of-expression =). The ! and = signs bracket the value to be placed into the variable specified to the left of the ! sign. (SETA (X value-of-I) value-of-expression) is set up and EVALed by LISP.

All variables are arrays. The logic here is simple. Since BASIC allows only the variables A, A0, A1, . . . A9, B,

```

30 dim x(10), y(10)
*** SYNTAX ERROR *** NOT AN OPERATOR,STRING, NUMBER, OR LETTER: DIM
=
;we have a new bug here. I changed something yesterday. sorry about that
=
lisp (+trace i-dim b-parse p-nxtelt)
$
(I-DIM B-PARSE P-NXTELT)
=
30 dim x(10)

-->> B-PARSE :: NIL
-->> P-NXTELT :: NIL
<<-- P-NXTELT :: DIM
-->> P-NXTELT :: NIL
*** SYNTAX ERROR *** NOT AN OPERATOR,STRING, NUMBER, OR LETTER: DIM
/1
LISP...
-
(edit b-parse)
:EDITING B-PARSE
:(LAMBDA NIL (PRG & & & GOODLINE: & &))
.
3 4 ?
:(COND ((AND (SETQ COMMAND (P-NXTELT)) (EQUAL SYNTYPE (QUOTE COMMAND))) (COND ((NOT
:(P-NXTELT))) ((IMplode (LIST (QUOTE P-) COMMAND))) (GO GOODLINE:)) ((OR (EQUAL SYNT
:NONSUBSCRIPT))) (P-LET) (GO GOODLINE:)) (T (P-ERROR (QUOTE "A LINE MUST START WITH
.
nx
:GOODLINE:
.
bk
:(COND (& & & &) (& & &) (T &)))
.
2 2 ?
:(COND ((NOT (MEMBER COMMAND (QUOTE (FILE GOSUB GOTO REM)))) (P-NXTELT)))
2 3 ?
:(QUOTE (FILE GOSUB GOTO REM))
.
2 ?
:(FILE GOSUB GOTO REM)
.
i dim before file 1 : ?
:(FILE GOSUB GOTO REM)
.
insert dim before gosub 1 : ?
:(FILE DIM GOSUB GOTO REM)
0
:(QUOTE (DIM FILE DIM GOSUB GOTO REM))
.
undo
:INSERT UNDONE
:(QUOTE (DIM FILE GOSUB GOTO REM))
.
ok
NIL

```

Figure 10. Using LISP: trace and edit.

B0, . . . B9, C . . . Z, Z0, . . . Z9, a general array VAR with 26 rows and 10 columns provides an easy definition for them. Dimensioned variables are also arrays defined with their own names. Thus we do not have to worry about the fact that X1 and X1(20) are two separate variables. The same structure is used for string variables, except they have a \$ sign attached (VAR\$ instead of VAR). Finally, an advantage of this structure is that the PARMS list is evaluated exactly the same way for variables of any type, system-defined functions or user-defined

functions. User-defined functions are parsed with dummy variables and inserted into the PARMS list, with the called variables replacing the dummy variables at run time. Some of the arithmetic functions are written in FORTRAN and attached to names in LISP. Such FORTRAN subroutines, though limited, allow for more efficient arithmetic work in LISP.

EDITOR

The EDITOR function is not very complex. A pro-

```
cms type bug1 lisp
```

```
(DEFINE (I-EVAL BUG
  ((FLAMBDA (ARGS) (PRINT 'ENTRY-TO-I-EVAL)
    (PRIN1 'ARGUMENTS:) (PRIN1 ARGS) (TERPRI)
    (TRACE SETQ UNCONS CCND CAR APPLY GET DEFINEA ))
  .(LAMBDA (RET) (PRINT 'EXIT-FROM-I-EVAL)
    (UNTRACE SETQ UNCONS COND CAR APPLY GET DEFINEA)
    (PRIN1 'VALUE:) (PRIN1 RET) (TERPRI)))))
```

```
=
;here we have bugged I-EVAL s that when we enter it, it will tell us that and
=
;put on a bunch of traces.
```

```
lisp (fetch bug1)
      (I-EVAL)
```

```
=
;see, now it is here.
```

```
=
runnh trace on
      10 S = SQR(.456)
ENTRY-TO-I-EVAL
ARGUMENTS: ((VAR 19 1 , ! SQR .4559995 % =))
```

```
+-->> SETQ :: (STACK NIL)
```

```
+<<-- SETQ :: NIL
```

```
+-->> COND :: (((NOT EXPRESSION) (GC ENDEVAL)))
```

```
+<<-- COND :: NIL
```

```
+-->> SETQ :: (ELEMENT (UNCONS EXPRESSION EXPRESSION))
```

```
+-->> UNCONS :: (EXPRESSION EXPRESSION)
```

```
+<<-- UNCONS :: VAR
```

```
+<<-- SETQ :: VAR
```

```
+<<-- SETQ :: (VAR)
```

```
+-->> COND :: (((NOT EXPRESSION) (GO ENDEVAL)))
```

```
+<<-- COND :: NIL
```

```
+-->> SETQ :: (ELEMENT (UNCONS EXPRESSION EXPRESSION))
```

```
+-->> UNCONS :: (EXPRESSION EXPRESSION)
```

```
+<<-- UNCONS :: 19
```

```
+<<-- SETQ :: 19
```

```
+<<-- SETQ :: (19 VAR)
```

```
+-->> COND :: (((NOT EXPRESSION) (GC ENDEVAL)))
```

Figure 11. Bugging a LISP function.

```

/1
LISP...

(edit p-expression)
:EDITING P-EXPRESSION
:(LAMBDA NIL (PROG & & LOOP &))
.
pp
:(LAMBDA NIL
:
: (PROG (OP)
: (COND ((NOT (P-TERM)) (RETURN NIL)))
: LOOP
: (COND ((NOT (MEMBER SYM
: '("+ " -" &))) (RETURN T))
: (T (SETQ OP SYM)
: (P-NXTELT)
: (COND ((NOT (P-TERM)) (RETURN NIL))
: (T (SETQ EXPVAL (APPEND EXPVAL OP))
: (GO LOOP))))))
:
(edit p-term)
:EDITING P-TERM
:(LAMBDA NIL (PROG & & LOOP &))
.
pp
:(LAMBDA NIL
: (PROG (OP)
: (COND ((NOT (P-FACTOR)) (RETURN NIL)))
: LOOP
: (COND ((NOT (MEMBER SYM
: '(: /))) (RETURN T))
: (T (SETQ OP SYM)
: (P-NXTELT)
: (COND ((NOT (P-FACTOR)) (RETURN NIL))
: (T (SETQ EXPVAL (APPEND EXPVAL OP))
: (GO LOOP))))))
:
(edit p-factor)
:EDITING P-FACTOR
:(LAMBDA NIL (PROG NIL & LOOP &))
.
pp
:(LAMBDA NIL
: (PROG NIL
: (COND ((NOT (P-PRIMARY)) (RETURN NIL)))
: LOOP
: (COND ((NOT (EQ SYM
: '***)) (RETURN T))
: (T (P-NXTELT)
: (COND ((NOT (P-PRIMARY)) (RETURN NIL))
: (T (SETQ EXPVAL (APPEND EXPVAL
: '***))
: (GO LOOP))))))
:
(edit= p-primary)
:EDITING P-PRIMARY
:(LAMBDA NIL (PROG & & UNARY & & & PAREN & & PERCENT & & GET_ARGS & &))
.
pp
:(LAMBDA NIL
: (PROG (OP)
: (COND ((EQ SYM
: '("(") (GO PAREN))
: ((OP (EQUAL SYM
: '("+")
: (EQUAL SYM
: '("-") (GO UNARY))
: ((NOT (MEMBER SYMTYPE
: '(CONSTANT NUMBER SUBSCRIPT NONSUBSCRIPT FUNCTION)) (RETU
: (T (SETQ EXPVAL (APPEND EXPVAL SYM))
: (P-NXTELT)
: (COND ((EQ SYM
: '%' (GO PERCENT))
: (T (RETURN T))))))
: UNARY
: (SETQ PLINE (CONS SYM PLINE))
: (COND ((NOT (SETQ SYM (P-NUMBER))) (P-ERROR "'INVALID OPERAND:" SYM)))
: (SETQ EXPVAL (APPEND EXPVAL SYM))
: (P-NXTELT)
: (COND ((EQUAL SYM
: '%' (GO PERCENT))
: (T (RETURN T))))
: PAREN
: (P-NXTELT)
: (COND ((NOT (P-EXPRESSION)) (RETURN NIL))
: (T (COND ((NOT (EQ SYM
: '(")") (P-ERROR "'RT PAREN NOT FOUND"
: (RETURN NIL))
: (T (P-NXTELT)
: (RETURN T))))))
: PERCENT
: (P-NXTELT)
: (COND ((NOT (P-EXPRESSION)) (RETURN NIL)))
: GET_ARGS
: (COND ((EQ SYM
: '(",") (P-NXTELT)
: (COND ((NOT (P-EXPRESSION)) (RETURN NIL))
: (T (SETQ EXPVAL (APPEND EXPVAL
: '(",")
: (GO GET_ARGS))))))
: (COND ((NOT (EQ SYM
: '(")") (P-ERROR "'RT PAREN NOT FOUND"
: (RETURN NIL))
: (T (P-NXTELT)
: (SETQ EXPVAL (APPEND EXPVAL
: '%))
: (RETURN T))))))

```

Figure 12. A pretty-print of the LISP function for parsing arithmetic expressions.

gram is defined as a list of line-number atoms in numerical order. The list begins with TOP and ends with EOF. The EDITOR keeps a pointer, *CURLINE, to the Nth item of the list. TOP and BOTTOM simply set the pointer to the second item or the next-to-the-last item on the list, respectively. UP and DOWN increase or decrease the pointer. When a command is interpreted, the line number is retrieved and its PLINE is printed. FIND and CHANGE are more complicated but require mainly list-searching and pattern-recognizing routines.

COMPILER

The initial revision of BAND AID did not include a COMPILER. The whole program was being interpreted, so it did not seem necessary to compile it before running. However, even though each line is parsed and represented in such a way that it is easy to interpret, there are some commands that depend on the correctness of other lines. The obvious ones are FOR-NEXT loops, GOTO commands, and GOSUB calls. It is also necessary to make sure that a number of parameters are initialized. The RUN options, TRACE, LIMIT, and DEBUG must

be set, multiple DATA lines combined into one list, variables set to zero, and files closed. Dimensioned variables are defined by the COMPILER. Perhaps its most important purpose is to make sure that loops are nested correctly and that NEXT statements are not encountered before their associated FOR lines are initiated.

INTERPRETER

We have already run through the interpretation of PARS for a LET line. The INTERPRETER moves from line to line calling the I-COMMAND function associated with each line. In most cases, these are simple functions—some are even null, since the COMPILER or the PARSER has already taken care of them. Rather more complicated commands are IF-THEN, CHAIN, OPEN and CLOSE (for files), PRINT and INPUT, FOR-NEXT loops, and, of course, expression evaluation. Because of LISP's poor arithmetic, we had to create a number of special function calls to FORTRAN. Being able to call FORTRAN functions is one of the more powerful features of the Michigan LISP. We can define

a call to a FORTRAN subroutine so that any complex function (e.g., Hashing, Fourier analysis, etc.) can be done from LISP. There are, however, two major and unfortunate limitations to the use of FORTRAN subroutines. One is that only compiled subroutines with text decks can be called, which prevents calling up a FORTRAN function that requires a load module. The reason for this limitation is that LISP attaches the text module to its internal data base. The second limitation is in passing of parameters. Any number of parameters can be passed, but they are passed by value only. The return value of the function is through one of the passed parameters. This means that no manipulation of matrices is possible, and it is almost impossible to pass arrays from LISP to FORTRAN. However, the usual EXP, LOG, SQR, SIN, COS, ATN, and power (XY) functions are FORTRAN calls. The INTERPRETER also contains functions for the string functions, which are relatively easy to define in LISP.

Utility Functions

In order to synthesize the previously described major components of the BANDAID system, it was necessary to define a large number of special-purpose functions. In many cases these were rather ad hoc in nature.

FURTHER DEVELOPMENTS

In an earlier paper, Millward, Mazzucchelli, Magoon, and Moore (1978) discussed the BANDAID system from a broad perspective, an overview that is still in effect. In this paper I have tried to provide a more detailed view of the program as it is now running and to make some points about its implementation. In the previous article, we mentioned developing a "schematic description of the student's program," a "knowledge network of BASIC language rules," a "semantic interpreter for natural language," and a "model of the student." Any implementation of these rather poorly understood components must wait for a great deal of further empirical research. Unfortunately, as I argued in my Presidential

Address to the National Conference on the Use of On-Line Computers in Psychology (Millward, 1979), a decent tutorial system will have to have some of these components if it is to be natural. Our goal is to build a system that is natural for the student. Progress toward this goal will be slow.

The BANDAID system now functions rather well as an interactive programming language. To make it a better tutorial system, we will have to add some ability to allow freer expression by the student. We will add a set of test programs and test items to insure that essential concepts are known. The addition of these facilities to the system must wait until we have more data on what tutors do in teaching students and on what kind of questions and problems students have with the present system. The Millward et al. (1978) paper spelled out the details of how we plan to monitor the interactions of a tutor and a student; we will report on the results of that work at some future time.

REFERENCES

- MILLER, L. Has artificial intelligence contributed to an understanding of the human mind? A critique of arguments for and against. *Cognitive Science*, 1978, 2, 71-109.
- MILLWARD, R. Teaching a computer to teach. *Behavior Research Methods & Instrumentation*, 1979, 11, 102-110.
- MILLWARD, R., MAZZUCHELLI, L., MAGOON, S., & MOORE, R. Intelligent computer-assisted instruction. *Behavior Research Methods & Instrumentation*, 1978, 10, 213-217.

NOTE

1. We are using an implementation of LISP obtained from Bruce Wilcox of the Mental Health Research Institute at the University of Michigan in Ann Arbor. This LISP was developed under an NSF Grant to Walter Reitman and the University of Michigan Computing Center. Since Michigan runs under its own time-sharing system, MTS, we had to convert it to run under Brown's version of CMS for the IBM 360. Ken Hoadly did that for us in his own free time, and we are indebted to him for his help. We have found the Michigan LISP an excellent research tool, even though we have not had all its features implemented here at Brown.