A MACHINE INDEPENDENT IMPLEMENTATION OF LOGO

by

Vincent S. Manis

B.Sc., University Of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

COMPUTER SCIENCE

We accept this thesis as conforming to the

required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

October 1973

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver 8, Canada

Date 10 OCTOBER 1973

Abstract

An implementation of the programming language Logo is described. Logo, a programming language somewhat like Lisp, is intended to teach the naive user the elements of programming and problem-solving, especially in symbolic programming applications such as graphics, natural language processing, musical composition, and the solution of elementary artificial intelligence problems. The system described here, BCLogo, is an attempt to build a portable implementation which would not be particularly sensitive to the computer on which it was run. The thesis describes both the manner in which the system appears to the user and the way in which the system was built.

Acknowledgements

During the preparation of this thesis, I have benefitted from the help of a number of people. First, my supervisors, J.E.L. Peck, Raymond Reiter, and D.A.R. Seeley, provided a great deal of support (especially moral) over the last few months. I am indebted for a large number of the good ideas here to Wallace Feurzeig and George Lukas, of Bolt, Beranek, and Newman, Inc., who spent a great deal of their time explaining why the original BBN implementation was the way it was. A number of other people, including Peter VandenBosch, Paul Friedman, Mark DuMont, and Miguel Alemparte, helped me to pinpoint errors in the system. The financial support of the National Research Council of Canada is gratefully acknowledged.

Table Of Contents

PART 1 INTRODUCTION

Logo is a language developed jointly by Bolt, Beranek, and Newman, Inc., and the Artificial Intelligence Laboratory at MIT, to aid novice computer users in programming computers. The language is very different from other naive user systems, such as Basic or APL, in that the language is not designed for numeric calculations, but for symbolic processing. Typical applications of the language are: graphics (especially "turtle geometry", a type of geometry using only the concepts of position and direction) [PAPE72], working with labyrinths [FEUR71], experiments in control theory, such as the processes of balancing a stick [PAPE71], poetry generation [PAPE71], and differential calculus [LUKA72]. The language has been described as a "baby Lisp" in that it, like Lisp, employs list processing, or symbolic, techniques--the basic datum is not the number, but the character string. The language emphasizes structured programming [DIJK72][FREE72], in particular the use of functional programming.

I am primarily concerned with encouraging the use of Logo in secondary schools. However, most secondary schools currently doing computing use a language such as BASIC. One of the primary reasons given for the widespread use of BASIC is that almost every computer manufacturer provides it--the same tired old justification given for Fortran. The availability of a Logo implementation which could be easily adapted to run on

a large number of different computers would go a long way toward answering the objection of BASIC's ubiquity.

The purposes of the work reported in this thesis are as follows:

- to construct a reasonably efficient implementation of Logo.
- to build a system to which new sets of primitive functions could easily be added. This facility will provide teachers with the ability to construct micro-worlds for their students to explore using Logo. One example of such a micro-world is that of control theory: the teacher may interface the Logo computer to a physical device such as a turtle or a cart, and then give the student a task such as moving the turtle along the edge of an object or balancing a stick. It must be possible to modify Logo to support such new devices easily.
- to construct a system which is quite "friendly" to to the naive user, in that each user can approach it at his own level, and the system (in particular the error messages) does not frighten beginners off. Many systems produce error messages such as "ERROR NUMBER K014", or "FILE NOT OPEN--EDIT LINE NOT ADDED". Such messages are not calculated to inspire confidence.
- to build a relatively machine independent implementation.

The last point needs some explanation: machine independence is a very elusive quantity. For example, we could have a four thousand line Fortran program written entirely in

ANSI Fortran, paying no attention to such questions as word size or operating system peculiarities. Yet if the particular computer we happen to be using doesn't have a Fortran compiler (many mini-computers have either no Fortran compiler, or only a Fortran interpreter), then we are really in trouble, because we now must translate our program into some other language; and nobody claims that Fortran is a "reasonable" language (ANSI has not set any standards on that). The method for machine independence which I have attempted is different from the "standard" approach often used: I have not tried to build something which would run instantly on every computer; instead, I tried to construct a system which would run on a number of computers, and was clearly enough expressed that it could be easily translated into an appropriate language for others. This Logo implementation is written in BCPL, a fairly machine-independent (but efficient) language designed especially for compiler writing. The language is such that algorithms can be clearly expressed, and nonnumerical problems such as interpreter design become extremely straightforward. The language has been implemented upon a number of different machines, including the Atlas 2 at Cambridge, the GE (now Honeywell) 645 and TX-2 at MIT, the PDP-10 at Bolt, Beranek and Newman, and the IBM 360/370. Nevertheless, it is eminently sensible to assume that there are a large number of users who might wish this system on their machine, but do not have a BCPL implementation. It is my hope that this system might serve as a model, in that at least the design does not have to be reattempted each time the system is moved to a new machine.

Part 1 Introduction

I have called this Logo implementation "BCLogo", which, if you think about it, is a rather bad pun. It currently operates on the UBC IBM 360/67 under the Michigan Terminal System. A version running within OS/360 will shortly be built.

PART 2 A USER'S GUIDE TO BCLOGO

{ The meaning doesn't matter if it's only idle chatter of a transcendental kind.
-- W.S. Gilbert, Patience }

This part of the thesis is a user's manual for the current system; it describes the facilities available to the Logo programmer, and the Logo environment.

2.1 Introduction

Logo is both a formal notation for describing algorithms and a practical system for programming computers. The fact that it is designed to work with symbols rather than numeric values is the prime factor which distinguishes it from languages such as Basic and Fortran. Many people also find that algorithms programmed in Logo are easier to read and understand than the same algorithms programmed in Basic.

This manual attempts to describe Logo from a somewhat elementary viewpoint. If you understand languages such as Lisp or APL you shouldn't have too much trouble with Logo. However, if your only exposure to computers has been with Basic or Fortran, you may have some trouble. In this case, please bear with me, and try not to let your previous programming experience interfere with your learning of a very different

language.

In order to learn a computer language, you must study two aspects of the language: the syntax--the rules regarding which programs are grammatically correct, and the semantics--the description of what a program does. One feature of Logo is that the syntax has a very subordinate role--there are very few syntax rules to learn. Correspondingly, the primary effort you will have to make lies in learning what the various features of Logo do, rather than how you write them down.

Many programming manuals start off with a section on notation; this one is no different. Please remember the following rule when reading the examples: the user's input to the computer is underlined; the computer's responses are not.

Normally, Logo operates in direct execution mode. This pompous phrase means that as you type a line into Logo, Logo obeys your command, rather than storing it away for later reference. You should think of Logo as a very fast, but not very smart, friend who is listening to what you say, and attempting to carry out your requests to the best of its abilities. However, Logo always sees things from its point of view; therefore, it will always do what you say, not what you mean.

2.2 Logo Data

{ The Roman Conquest was, however, a Good Thing.
-- Sellar And Yeatman,
1066 And All That }

One of the most important constraints upon any programming language is the choice of data type that may be manipulated. For example, Fortran is very good at manipulating numbers, and correspondingly poor at processing textual material[1] Logo excels at processing text, and is not particularly suited to numeric calculation. Nonetheless, for a reasonably small calculation (for example, any program written by a high-school student in Basic), Logo will work quite well.

---

[1] one interesting consequence of Fortran's rise to dominance in the computing world is that there is a breed of Fortran programmers who are good at text processing; they write subroutines for all the other Fortran programmers, who regard even the simplest processing of character text as a mysterious experience to be avoided wherever possible.

2.2.1 Words

{ In the Beginning was the Word
-- The Gospel According To John }

The basic object with which Logo deals is the word. A word is any string of letters and digits. As examples, consider EGGPLANT, 1000%, THIS.IS.A.LONG.WORD....., and 45. A word can have as few as one and as many as 255 characters in it.

2.2.2 Sentences

Once you have created a set of words, you will probably want to tell some sort of story. In English, we combine words into sentences; Logo does the same: a Logo sentence is simply a sequence of words separated by blanks. For example,

```
THE EGGPLANT ATE THE AARDVARK
1 AND BUT !! 4 D9NUBT6L GJDDJD
```

are sentences. You should notice that we don't end Logo sentences with periods.

2.2.3 Lists

We can get even more extravagant, by joining sentences together. For example, we could have a thing made by joining three other things together: the first and third elements could be sentences and the second a word. We call any thing made by joining words or sentences together a list. (What relation do sentences bear to lists?) Lists provide you with a great deal of power--you can create extremely complex things with lists for example, you can store a matrix as a list--a matrix can be viewed as a list each of whose elements is a sentence. The matrix

$$\begin{bmatrix} 1 & 4 & A \\ 2 & 5 & B \\ 3 & 6 & C \end{bmatrix}$$

Might be represented as

```
[ [1 4 A] [2 5 B] [3 6 C] ]
```

or as

```
[ [1 2 3] [4 5 6] [A B C] ]
```

depending on whether we are primarily interested in rows or columns. Lest all this power run to your head, I must point out that for reasonable problems no great utility has been found for lists which are not sentences (there are other

methods of creating complicated objects which are easier to understand and debug), and, therefore, you should probably stay with words and sentences.

2.2.4 Quoting Things

```
{ Ah yes! I wrote the "Purple Cow"--
  I'm sorry now I wrote it!
  But I can tell you anyhow
  I'll kill you if you quote it!
      -- F.G. Burgess, Cinq Ans Apres }
```

So far, we have only discussed the kinds of abstract things you can use when working with Logo. Since Logo is not only a formal notation, but also a computer language, you must have some means for describing Logo things to your computer--you must be able to quote a thing to the computer. You may use a number of conventions for specifying the data:

- surround a word with quotes: "HELLO" "1000%" "4"
- surround a sentence with quotes: "A MESS OF POTTAGE"
- surround a list with brackets[1]:

  [HE ATE [THE LAST] EARTHWORM]

As a special concession to our historical conventions, you can enter a word composed entirely of digits without the surrounding quotes. Logo will assume that the quotes were

---

[1] it is possible for your system programmer to create a version of the Logo processor which uses other characters in place of the brackets. If you are using such a version of the system, please (mentally) substitute the chosen characters for the brackets throughout this manual.

present--thus it will treat 45 and "45" identically. Since this feature only applies to numbers, it can lead to some confusion--at least when you are learning Logo. Therefore, I would recommend that you avoid this feature, and always use quotes to surround words.

2.2.5 The Empty Thing

> { He finds it empty, swept, and put in order
> -- The Gospel According To Matthew }

One of the more amusing things about Logo is the empty thing. This unique creature is neither word nor sentence (nor list). It plays the role in Logo that zero plays in arithmetic. To make a joke, the empty thing is written as follows:

but, when talking about it to Logo, we surround it with quotation marks:

""

Empty is a great deal of fun: it is an object which isn't there--when you combine empty with another object, you end up with the original one. If you were to repeatedly remove

something from some word, sentence, or list, you would eventually end up with empty. Finally, note that every Logo thing has an infinite supply of empties hidden in it, just as every number has an infinite supply of zeroes hidden within it.

2.2.6 Some Frills

When you quote a Logo thing to the computer, excess blanks will be ignored. For example,

" A B C " is equivalent to "A B C", and
" " is equivalent to ""

If you really want to enter a blank as a literal character, you must type a sharp (#). Thus,

"FLEE,#HE#CRIED"

is one single word, not a sentence, which contains sharps, not blanks. Nonetheless, it will print out as

FLEE, HE CRIED

Note that "A###B" is <u>not</u> equivalent to "A#B".

2.3 The Syntax Of Logo

2.3.1 Logo's Syntax Rules

Every language has a set of rules which define those things that are valid in that particular language. Logo is no exception--you must follow Logo's rules very carefully, or Logo will not do what you want. The syntax of a language is simply the set of rules which define the things you can say in that language.

To understand Logo's syntax, we must recall that a computer language describes essentially two different kinds of things: the data to be operated upon, and the operations to be applied to the data. Note that these aspects are interdependent--it makes as little sense to tell the computer what to do (while neglecting to inform it of what data are to be used) as it does to merely give the computer some data, with no instructions as to what needs to be done.

The Logo syntax is extremely simple: almost every instruction is of the form:

<operation> <datum 1> <datum 2> ... <datum n>

In other words, the action to be performed always precedes the listing of the data to be used. While that is not quite all of

the syntax of Logo, it will help you with most of the instructions you might want to write.

At this point, we must examine the kinds of instructions one might write. One kind is the command, which is an order to do something ("Open your book to page 57."). Contrast this with an operation, which answers a question ("when did the Venomous Bead live, and what did he write?"). The essential difference is that while the result of a command is some action, the result of an operation is itself some sort of datum. This means that an input to an operation (one of the data listed after the name of the operation) could itself be the output, or result, of another operation. The same phenomenon occurs in English ("Who wrote 'Utopia'?"; "How was the person who wrote 'Utopia' executed?"); therefore, it should hardly be surprising to you in Logo.

Suddenly the Logo instructions become more interesting--you can create very extravagant commands which cause large numbers of operations to be executed. Nevertheless, the syntax rules are very simple indeed:

- for both operations and commands, the instruction is written with the name of the desired action, followed by the data that the action needs.
- the input to an operation or command may itself be the output of another operation (remember, commands don't output).
- a statement (a complete instruction to Logo) must always be a command.

2.3.2 Some Examples

At this point, let's look at some examples; but first, we need some actions to talk about.

- PRINT x is a command which causes its input to be typed on the user's console.
- SUM x y outputs the sum of the two numbers 'x' and 'y'.
- WORD x y outputs a word composed of the characters of 'x' followed immediately by those of 'y'.
- IGNORE x is a command which takes its input and does absolutely nothing with it.

PRINT "HELLO" causes HELLO to be typed on the user's console.

PRINT SUM "1" "2" causes 3 to be typed on the user's terminal.

PRINT SUM "4" SUM "5" WORD "6" "7" types out 76.

IGNORE SUM "1" "2" causes 3 to be computed, and absolutely nothing done with the result. Nothing is typed out.

SUM "1" "2" causes Logo to complain
"YOU DIDN'T TELL ME WHAT TO DO WITH 3."

2.3.3 Noise Words And Comments

Logo allows you to insert "noise words" into your commands in order to help clarify their meaning. These words are ignored by Logo, and may be inserted or left out, as you wish. The noise words permitted are:

- OF may be inserted between a function name and its inputs. For example:

```
PRINT SUM OF "1 2"
```

- AND may be inserted between two inputs of a function:

```
PRINT SUM "1" AND "2"
PRINT SUM OF "1" AND SUM OF "2" AND "3"
```

In addition, you may surround an input with parentheses, in order to remind yourself that it is one input. For example:

```
PRINT (SUM OF "1" AND "2")
PRINT (SUM OF ("1") AND (SUM OF "2" AND "3"))
```

Note that

```
PRINT SUM OF ("1" AND "2")
```

Is invalid, since the parentheses do not surround a single input.

It is sometimes convenient to annotate your commands. You may place a comment (to yourself) in any command by surrounding it with semicolons. If the comment is the last

thing on the line, the trailing semicolon may be elided.

```
PRINT :THE: SUM OF :INPUTS: "1" AND "2"
PRINT WORD OF "A" AND "B" :SHOULD PRINT "AB".
```

2.4 Defining Procedures

2.4.1 Procedures

If all that Logo allowed you to do were to use the procedures which the designers of the language saw fit to include, then it wouldn't be much good. The chief virtue of Logo, in fact, is that it permits you to make new procedures to fit your needs. To take one of Gerald Weinberg's examples [WEIN71], you cannot expect your language to have a canned procedure for generating Buddhist prayers, or even for generating prayers for any denomination (for example, PRAY 143 "MALACANDRIAN"). However, if the language allows you to write a PRAY procedure, and then use it in such a way that you don't have to worry whether it's built-in or not, then you can regard PRAY as built-in. Put another way, Logo by itself provides you with a battery of resources; if none of the resources exactly suits your needs, you can easily create one which does. Then, you would have your own version of Logo, which had all of the features of Logo, plus the one extra function you need to solve your problem.

This viewpoint is in sharp distinction from that of, say, Basic, in which the emphasis is on writing a problem to do something, and you never name the components of the program according to what they do.

The basic things you must do when defining a procedure are

- think up a name that is very descriptive of the procedure. If you do this first, you may save yourself a lot of trouble later on, in that now at least you have some sort of identification for the problem you are trying to solve.
- decide on what inputs the procedure is to have, and whether the procedure is to be a command or an operation. If it's to be an operation, decide what the output is to be.

At this point, you have defined a "black box": an object which has clearly defined behavior (on given inputs A, B, and C, the result is D); you also have a name for the box, which helps you when talking about the function to others.

- decide upon the algorithm to be used.

2.4.2 Simple Procedures

Let us take a very simple case: we want a procedure which prints out "HELLO MORTIMER". We can define this procedure as follows:

```
TO GREETMORTIMER
10 PRINT "HELLO MORTIMER"
END
```

Notice several things: first, we typed in a "title line", consisting of the word 'TO' followed by something which is highly unlikely to be already defined in Logo. At this point the computer is prepared not only to honour commands which are typed in; it also allows you to type in lines beginning with a number. If you type such a line, you will observe no immediate result. The computer will merely store your line away, noting that it is line 10 (in this case) of the function you are currently working with. You can type as many lines as you like into a function; this example just has one line (note that if you ever have a function longer than about 15 lines, you will probably find it both unreadable and undebuggable). The command END causes Logo to note that you have finished editing the current function; it will respond with

```
GREETMORTIMER DEFINED.
```

Now you are in a position to try out your new procedure:

```
GREETMORTIMER
HELLO MORTIMER
GREETMORTIMER
HELLO MORTIMER
```

2.4.3 Inputs

After you have run GREETMORTIMER a few thousand times the fun will begin to pall. What we would really like is a function which will greet anybody at all. Let's look at this one:

```
TO GREET :PERSON:
10 PRINT SENTENCE "HELLO" :PERSON:
END
GREET "ELMER"
HELLO ELMER
GREET "FERDY"
HELLO FERDY
```

The difference here is in the title line. Note that we have placed the word PERSON after the name of the procedure. By enclosing PERSON in colons, we are indicating to Logo that PERSON is a placeholder, standing for any input we care to hand to GREET. (incidentally, ':PERSON:' is pronounced "dots PERSON" by Logo cognoscenti).

You may define a procedure with as many inputs as you

want; for example

```
TO GRUMBLE :WHY: :HOWMANY:
10 PRINT "I AM SO UNHAPPY"
20 PRINT SENTENCE "BECAUSE" :WHY:
30 PRINT WORD :HOWMANY: WORD :HOWMANY: :HOWMANY:
END
GRUMBLE "I LOST MY KIWI" "GREEN"
I AM SO UNHAPPY
BECAUSE I LOST MY KIWI
GREENGREENGREEN
GRUMBLE "WOLVES ATE MY PTARMIGAN" "HA"
I AM SO UNHAPPY
BECAUSE WOLVES ATE MY PTARMIGAN
HAHAHA
```

2.4.4 Outputting Results

You can make a procedure behave as an operation by using the OUTPUT command. For example:

```
TO TRIPLE :X:
10 OUTPUT WORD :X: AND WORD :X: AND :X:
20 PRINT "HELLO, I AM TRIPLE"
END
PRINT TRIPLE "ORK"
ORKORKORK
```

OUTPUT causes its input to be returned as the value of the procedure from which it was called. Execution then stops. Therefore line 20 of TRIPLE had no effect--line 10 was the last to be executed.

2.4.5 Recursion

The procedures we have looked at so far don't do very much. What we want is a mechanism for repeating a procedure over and over again. This way, we don't have to write a long list of things for the computer to do; instead, once we have told Logo how to do something, we can have Logc repeat that action as many times as we want. One mechanism we can use is called recursion. It consists of having a procedure which calls itself. Consider

```
TO CACKLE
10 PRINT "HA"
20 CACKLE
END
```

This procedure executes line 10, thus printing out HA. Next it executes line 20, which requests another CACKLE. So it does it. Line 10 prints out HA, and then line 20 requests another CACKLE. So it does: line 10, line 10, line 10, ...

2.4.6 Conditions

Now that we have created a maniac computer which is gibbering "HA HA HA ...", we want to bring it under control. One way is obviously to press the interrrupt button; however, this solution lacks a certain amount of elegance (we can call this the ad-hoc, solution). Another method is to include within the program a command to stop when it has chortled enough. In other words, the computer must make a decision.

What kind of decision can a computer make? It can check to see whether a number is zero, whether a word is empty, or whether one word is equal to another. (There are a large number of other decisions which can be made, but these are enough for the moment). All of these decisions yield an output of either TRUE or FALSE--because of this, they are known as predicates. A predicate is any operation which always outputs either TRUE or FALSE. For example, the Logo operation IS is a predicate which takes two inputs, each of which is any Logo thing. It outputs TRUE if the two inputs are equal, and FALSE otherwise. Therefore, IS "ABC DEF" "ABC DEF" is TRUE, whereas IS "ABD CEF" "ABC DEF" is FALSE. If we were to feed the output of IS as an input to PRINT, we would know whether the two inputs were equal or not. However, the whole point of predicates is to have the computer make the decision. Therefore, we use TEST, which is a command which takes as input a truthvalue--TRUE or FALSE. TEST doesn't appear to do very much--it simply stores its input away in a special location called the truth flag. We can then use a number of other commands to interrogate the truth flag. For example, consider the following sequence:

```
TEST IS "APPLE" "ORANGE"
IFTRUE PRINT "RASPBERRY"
IFFALSE PRINT "PEACH"
PEACH
TEST IS "APPLE" "APPLE"
IFTRUE PRINT "PEAR"
PEAR
IFFALSE PRINT "CHERRY"
```

The primary use of TEST is in procedures. For example,

```
TO LOOKAT :WORD:
10 TEST IS :WORD: "STOP"
20 IFTRUE PRINT "OK"
30 TEST IS :WORD: "GO"
40 IFTRUE PRINT "RIGHT!"
50 IFFALSE PRINT "NO!!!"
END
LOOKAT "STOP"
OK
LOOKAT "GO"
RIGHT!
LOOKAT "PSITTACOSIS"
NO!!!
```

2.4.7 More Recursion

Now we can look at some real examples of recursion. Our first example is a procedure called GIGGLE, which takes a parameter which indicates how many times to laugh.

```
TO GIGGLE :N:
10 TEST IS :N: 0
20 IFFALSE PRINT "HA"
30 IFFALSE GIGGLE DIFFERENCE :N: 1
END
GIGGLE DEFINED.
GIGGLE 5
HA
HA
HA
HA
HA
```

GIGGLE's first step is to se whether it has done enough. If so, the remainder of the procedure is bypassed by the IFFALSEs. Otherwise, we do some work, and then call GIGGLE again with an input which is 1 less than it was the previous time.

We can use the TRACE function to see what GIGGLE is doing.

```
TRACE GIGGLE
GIGGLE 3
GIGGLE OF "3"
HA
  GIGGLE OF "2"
HA
    GIGGLE OF "1"
HA
      GIGGLE OF "0"
      GIGGLE STOPS
    GIGGLE STOPS
  GIGGLE STOPS
GIGGLE STOPS
```

For a second example, consider a procedure which reverses a word:

```
TO REVERSE :W:
10 TEST IS :W: ""
20 IFTRUE OUTPUT ""
30 OUTPUT WORD REVERSE (BUTFIRST :W:) FIRST :W:
END
REVERSE DEFINED.
TRACE REVERSE
PRINT REVERSE "MIMSY"
 REVERSE OF "MIMSY"
  REVERSE OF "IMSY"
   REVERSE OF "MSY"
    REVERSE OF "SY"
     REVERSE OF "Y"
      REVERSE OF ""
      REVERSE OUTPUTS ""
     REVERSE OUTPUTS "Y"
    REVERSE OUTPUTS "YS"
   REVERSE OUTPUTS "YSM"
  REVERSE OUTPUTS "YSMI"
 REVERSE OUTPUTS "YSMIM"
YSMIM
```

FIRST outputs the first element of its input; BUTFIRST outputs all but the first element of its input. Thus, for example, FIRST "HELLO" is "H", while BUTFIRST "HELLO" outputs

"ELLO".

2.4.8 Variables

By now, you must be getting a bit suspicious about all these names enclosed in colons. The technical name for a word enclosed in colons is "variable". A variable simply is something that can vary: for example, in our procedure GREET, today :PERSON: might be "FRED" and tomorrrow it might be "MIKE". A variable is simply the name of a pigeonhole in which any Logo thing can be stored.

The simplest kind of variable is the input to a procedure. Inputs are set to the value given when you call the procedure. However, since a number of procedures might use the same name for an input, Logo does a bit of bookkeeping for you. If you have a procedure MYPROC, with an input named :Y:, then when you call MYPROC, Logo will save the previous value of :Y:, and set :Y: to the value specified as the input to MYPROC. When MYPROC finishes, :Y: will be set to its previous value. Since the inputs to a procedure are used only in the locale of that procedure, we call them "local variables".

We often call the value associated with a variable the "Thing" of the variable.

Often we would like to use a variable within a procedure, even though that variable is not an input to the procedure. For example,

```
MAKE "S" "THIS IS A VERY LONG SENTENCE"
TO PR :V:
10 PRINT :V:
20 PRINT :S:
END
PR "HELLO"
HELLO
THIS IS A VERY LONG SENTENCE
```

MAKE is a command which sets the Thing of its first input to be its second input. This procedure uses the value of S, even though S is not an input; S is therefore called "global". This usage is quite permissable, and allows a great deal of convenience in that not everything a procedure uses needs to be an input.

However, it is possible to get into trouble, as shown in the following example:

```
MAKE "X" 10
TO MESS :Y:
10 PRINT :Y:
20 MAKE "X" SUM :Y: 1
30 PRINT :X:
END
MESS DEFINED
MESS 3
3
4
PRINT :X:
4
```

In this example, we (unintentionally) changed the "global" X. What we would like to do is to "protect" the global X, in in the same way that an input named Y protects the global Y (if there is one). We do this by declaring X "local", with a title line like

TO MESS :Y: USING :X:

This declares that MESS will "use" X; we want the global X to be protected. Therefore, when we start executing MESS, Logo will save the previous value of X (and, for that matter, Y). When MESS stops, the previous values will be restored.

2.4.9 Frills

Sometimes it is convenient to halt execution of a procedure before the END line. The STOP command causes execution of the procedure from which it was called to cease immediately, and return to its caller. STOP is used for those procedures which return no values in the same way that OUTPUT operates for those procedures which do return values.

The command GOTOLINE takes one input which must evaluate to a number. Control in the current procedure is transferred to the line identified by its input.

```
TO FRILL
10 MAKE "N" "DOG"
15 PRINT "HELLO"
20 TEST IS :N: "CAT"
30 IFTRUE STOP
40 PRINT "CANINES HATE FELINES"
50 MAKE "N" "CAT"
60 GOTOLINE 20
END
FRILL
HELLO
CANINES HATE FELINES
```

2.4.10 Another Example

As a last example, we give a solution to the famous Towers of Hanoi puzzle. This puzzle is played with three pegs, and a number of discs of differing sizes. One starts with the discs on the "source" peg, arranged in order of size, with the smallest disc on top. The object is to move the entire stack onto the "destination" peg, subject to the following restrictions:

- only the top disc on a stack may be moved at any time.
- only one disc at a time may be moved.
- no disc may be placed on a smaller one.

The following program implements the most famous solution: the recursive one in which a problem of order N is solved in three steps:

- solve the problem of order N-1 in order to move all but the bottom disc from the "source" to the "intermediate" peg.
- move the bottom disc to the "destination" peg.
- move all of the discs on the "intermediate" peg to the "destination".

```
TO HANOI :N: :S: :I: :D:
10 TEST IS :N: 0
20 IFTRUE STOP
30 HANOI (DIFFERENCE :N: 1) :S: :D: :I:
40 PRINT S "MOVE DISC" S :N: S "FROM" S :S: S "TO" :D:
50 HANOI (DIFFERENCE :N: 1) :I: :S: :D:
END
HANOI DEFINED.
HANOI 4 "SOURCE" "INTERMEDIATE" "DESTINATION"
MOVE DISC 1 FROM SOURCE TO INTERMEDIATE
MOVE DISC 2 FROM SOURCE TO DESTINATION
MOVE DISC 1 FROM INTERMEDIATE TO DESTINATION
MOVE DISC 3 FROM SOURCE TO INTERMEDIATE
MOVE DISC 1 FROM DESTINATION TO SOURCE
MOVE DISC 2 FROM DESTINATION TO INTERMEDIATE
MOVE DISC 1 FROM SOURCE TO INTERMEDIATE
MOVE DISC 4 FROM SOURCE TO DESTINATION
MOVE DISC 1 FROM INTERMEDIATE TO DESTINATION
MOVE DISC 2 FROM INTERMEDIATE TO SOURCE
MOVE DISC 1 FROM DESTINATION TO SOURCE
MOVE DISC 3 FROM INTERMEDIATE TO DESTINATION
MOVE DISC 1 FROM SOURCE TO INTERMEDIATE
MOVE DISC 2 FROM SOURCE TO DESTINATION
MOVE DISC 1 FROM INTERMEDIATE TO DESTINATION
```

S is a Logo synonym for SENTENCE. Tracing HANOI may help you understand it.

2.5 Editing And Workspace Management

Once you have defined a function, you may change that definition by using the EDIT command. For example, if you were to do

```
TO CHIRP
10 PRINT "CHIRP CHIRP"
END
```

Now suppose you wish to modify line 10. You could execute

```
EDIT CHIRP
```

Now you are in editing mode; you can insert or delete lines, and you can SHOW the active function, just as if you were defining CHIRP again. For example,

```
EDIT CHIRP
20 PRINT "HELLO THERE"
END
CHIRP
CHIRP CHIRP
HELLO THERE
```

If you are in editing or definition mode, you can use the SHOWLINE command to display a specific line of your function. For example, assume that we're editing CHIRP.

```
EDIT CHIRP
SHOWLINE 10
10 PRINT "CHIRP CHIRP"
```

Sometimes it is convenient to display the entire definition of a function. If you are editing a function, the command SHOW will display the definition of that function.

```
EDIT CHIRP
SHOW
TO CHIRP
10 PRINT "CHIRP CHIRP"
20 PRINT "HELLO THERE"
END
```

At any time (even if you are not currently editing anything), you may display the definition of a function X by typing SHOW X

```
SHOW CHIRP
TO CHIRP
10 PRINT "CHIRP CHIRP"
20 PRINT "HELLO THERE"
END
```

To erase the definition of a function, execute

```
ERASE CHIRP
CHIRP ERASED.
CHIRP
I DON'T KNOW HOW TO CHIRP.
```

In order to allow you to develop a program over a long period of time, Logo allows you to save the current workspace (consisting of your procedures, the truthflag, and the values of the variables), and reload it at some later time. Workspaces are saved in files ; a file is an area of disc storage to which you have given a name. If you have done some work and wish to save it, execute

SAVE filename

Where 'filename' is the name you wish to give the file. If 'filename' already exists, you will be asked to confirm that you really did want to save the file under that name, since you might accidentally write over a valuable workspace.

One very common use of these facilities is the following:

- start a session off by GETting a workspace.
- define or edit functions.
- save the resulting workspace (the material which was gotten, plus what you have just done), back into the file you got the original workspace from.

To retrieve the information you have SAVEd, you can execute

GET filename

This command adds the material stored in the file to what is currently in the workspace.

2.6 Data Structures

Note: this section is probably of no interest to the beginning Logo programmer; it is intended for the edification of those familiar with the list-processing facilities of, for example, Lisp, Algol W, or PL/I.

Although Logo is a very simple language, you can use it for many list-processing problems. This section is intended to explain how to accomplish this.

A list-processing algorithm is one which builds its data structure from elementary objects, or nodes, together with housekeeping information which is used to connect the nodes together in some order. The housekeeping data are called pointers, or node names. It doesn't matter what they are, as long as we can always rely upon a given pointer to "point our way" to the same node. In Fortran we can use integers, in Algol W or PL/I machine addresses, and in Logo words. In addition, we must have a "null", which doesn't point anywhere.

Suppose we had a collection of objects which we wanted to

build into a chained data structure. One way would give us a data structure which would look like that of figure 2-1. The first segment of each node is the object under consideration, and the second segment points to the next node in the structure.

In Logo terms, a pointer to a node N can simply be a word whose Thing is N. For example, if we were to execute

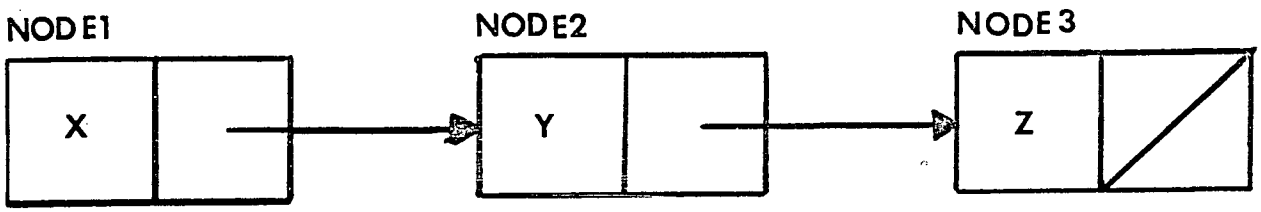
```
MAKE "SUBSCRUBIAL" "NUGATORY"
```

Then we could say that SUBSCRUBIAL points to NUGATORY. Therefore, if we have a pointer P, we can get to the object it refers to by simply executing THING OF "P". Our null can be the empty thing, as it can't be used to name anything, anyway.

As a list processing example, suppose we have the data structure of figure 2-1. We can print it out by executing

```
TO PSTRUC :P:
10 TEST EMPTYP :P:
20 IFTRUE STOP
30 PRINT FIRST :P:
40 PSTRUC THING OF LAST OF :P:
END
```

To create such a data structure, we must have a supply of names. We could, in principle, use any Logo word as a pointer; however, in order to avoid confusion between pointers and other names (such as those for inputs), we should use "strangely spelled" names. One sequence which will admirably suit our purposes is $$1, $$2, $$3, ... Failure to differentiate between names used for pointers and other names is a common but difficult to interpret bug. We can have a



Each node in the data structure consists of:

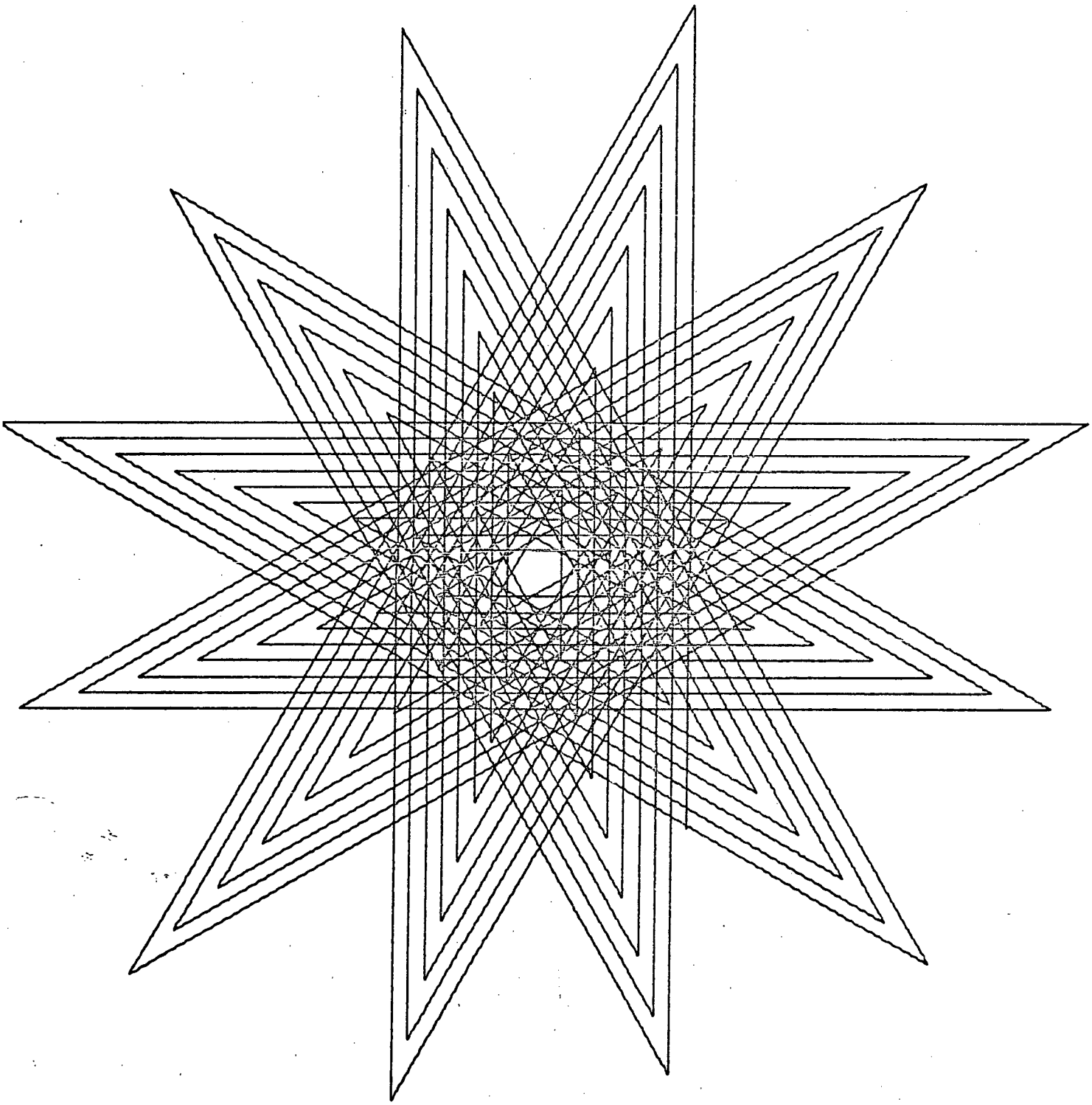
* a datum, and

* a pointer to the next node in the structure.

The structure could have been built by executing:

```
MAKE "NODE1" "X NODE2"
MAKE "NODE2" "Y NODE3"
MAKE "NODE3" "Z (empty)"
```

Figure 2-1
A Data Structure.



TURTLE POWER!

procedure POINTER, which returns a new pointer each time it is called.

```
TO POINTER
10 TEST EMPTYP :PCOUNTER:
20 IFTRUE MAKE "PCOUNTER" 0
30 MAKE "PCOUNTER" SUM "PCOUNTER" 1
40 OUTPUT WORD "$$" AND :PCOUNTER:
END
```

Lines 10 and 20 are for initialisation: the first time POINTER is called, PCOUNTER will be unassigned; thus these lines will set it to 0.

Now we can create our data structure:

```
TO CREATE USING :P: :Q:
10 TYPEIN "P"
20 TEST IS :P: "/END/"
30 IFTRUE OUTPUT ""
40 MAKE "Q" POINTER
50 MAKE :Q: SENTENCE :P: CREATE
60 OUTPUT :Q:
END
```

CREATE reads in a datum. If it is the special sentinel /END/, we know we have reached the end of the list being built. Otherwise, we make an a new pointer, and set it to a new node consisting of the datum we just read in, joined with the pointer produced by another call to CREATE. Here's a trace of CREATE.

```
TRACE CREATE
MAKE "MYLIST" CREATE
 CREATE OF
A
  CREATE OF
B
   CREATE OF
C
    CREATE OF
/END/
    CREATE OUTPUTS ""
   CREATE OUTPUTS "$$1"
  CREATE OUTPUTS "$$2"
 CREATE OUTPUTS "$$3"
```

2.7 The Logo Functions

The intent of this section is to give you an idea of the various Logo functions, and a vague idea of what they do. The functions are listed by classes. For a particular function, a complete description will be found in section 2.8.

2.7.1 Text Processing

- FIRST, LAST, BUTFIRST, and BUTLAST allow you to take apart words, sentences, or lists.
- COUNT outputs a number indicating how many elements are in its input.
- WORD, SENTENCE, and LIST allow you to create new things.

2.7.2 Predicates

- IS is TRUE when its two inputs are equal.
- WORDP is TRUE when its input is a word.
- SENTENCEP is TRUE when its input is a sentence or list.
- BOTH is TRUE when both of its inputs is.
- EITHER is TRUE when either of its inputs is.
- NOT is TRUE when its input is false.
- NUMBERP is TRUE when its input is a numeric word.
- GREATERP is TRUE when its first input is numerically greater than its second.
- LESSP is TRUE when its first input is numerically less than its second.
- EQUALP is TRUE when its inputs are numerically equal.
- ZEROP is TRUE when its input is numerically equal to 0.

2.7.3 Arithmetic

- SUM, DIFFERENCE, and PRODUCT do the familiar operations of addition, subtraction, and multiplication.
- DIVISION outputs a sentence of the quotient and remainder from a division. QUOTIENT and REMAINDER can be used to compute just one or the other result.
- RANDOM outputs a random digit; RANDNO outputs a random number within a specified range. SEED can be used to examine or change the random number seed.

- MAXIMUM and MINIMUM select the maximum or minimum of two numbers.

2.7.4 Functions

- TO allows you to define a new function.
- EDIT allows you to change the definition of an existing function.
- END signals that you have finished defining or changing a function.
- SHOW displays the definition of a function.
- SHOWLINE displays a particular line of a function.
- ERASE expunges the definition of a function.
- ERASELINE allows you to delete a particular line of a function.
- TITLE changes the title line of a function.
- OUTPUT causes the currently executing function to return OUTPUT's input as its value.
- GOTOLINE transfers control within the currently executing function.
- STOP terminates execution of the currently executing function.
- EXIT types its input, and then behaves as if an error had occurred.

2.7.5 Conditionals

- TEST sets the truth flag from its input.
- IFTRUE executes a statement only if the truth flag is TRUE.
- IFFALSE executes a statement only if the the truth flag is FALSE.

2.7.6 Input/Output

- REQUEST reads in a line of data, and outputs a list representing that line.
- TYPEIN takes an input; it is equivalent to "MAKE input REQUEST".
- TYPE prints its input onto the user's console.
- PRINT behaves as TYPE does, but follows the printing by typing a carriage return.

2.7.7 Timing

- CLOCK outputs the amount of time you have used so far.
- RESETCLOCK sets the CLOCK value to 0.
- TIME outputs the time of day in the form of a sentence "hh mm ss", in which hh is the hour (in the 24-hour clock), mm the minute, and ss the second.
- DATE outputs the date as a sentence of the form "ww yyyy mm dd", in which ww is the name of the day of the week, yyyy is the year, mm is the name of the month, and dd is the day of the month. For example, "TUESDAY 1973 OCTOBER 9".

2.7.8 System Control Functions

- GOODBYE gets you out of Logo.
- LOGOUT gets you out of Logo, and in addition, hangs up the telephone line to the computer.
- GET allows you to add material from a file to the programs and variables with which you are currently working.
- SAVE stores your current workspace in a file.
- TRACE causes a specified function to be monitored for debugging purposes.
- UNTRACE turns off debug monitoring for a function.

2.7.9 Variables

- MAKE changes the value of a specified variable.
- THING returns the thing of a variable.

2.7.10 Program Manipulation Functions

- DO takes as its input a thing representing a Logo statement; it executes that statement.
- LINES and TEXT are used to manipulate procedures: if X is the user procedure

```
TO X :A:
10 PRINT "HELLO"
20 PRINT :A:
END
```

Then LINES "X" returns the sentence "10 20"--the sentence of line numbers of a procedure. TEXT "X" 20 returns the sentence "20 PRINT :A:". These two functions allow your program to go through a function, scanning for something, or doing some systematic alteration. In order to allow you to get the title line, line number 0 references the title: TEXT "X" 0 is "TO X :A:". Note that DO can be used to insert a line: DO "10 PRINT 1". |se('2.7.11 miscellaneous functions')

- IGNORE takes one input and does absolutely nothing with it.

2.8 A Glossary Of Logo Functions

This section describes the set of Logo functions currently available. The functions are listed in alphabetical order. Some functions have synonyms (for example, PRINT and P have the same effect); the synonyms of a given function name are listed after the function name.

While this list looks very imposing, you should not take it too seriously: nobody really expects that a beginning Logo student should really learn fifty functions off by heart. When teaching Logo, one should start off with a subset of about 15 functions; the exact set would vary according to the needs of the student. One such set would be:

FIRST, BUTFIRST, WORD, SENTENCE, TEST, IS, IFTRUE, IFFALSE, TO, EDIT, SHOW, ERASE, END, GOODBYE, OUTPUT

However, you can tailor the exact set to your needs.

BOTH	(2-input operation) outputs TRUE if both of its inputs are TRUE; otherwise it outputs FALSE.
BUTFIRST BF	(1-input operation) outputs a thing made up of all but the first element of its input. Generates an error if given empty.
BUTLAST BL	(1-input operation) outputs an thing made of all but the last element of its input. Generates an error if given the empty thing.

CLOCK	(0-input operation) outputs the amount of time since the last call to RESETCLOCK. If RESETCLOCK has not been called, outputs the amount of time since Logo was started.
COUNT	(1-input operation) outputs a number indicating how many elements are in the input. COUNT of a word is the number of characters in it. COUNT of a sentence is the number of words in it. COUNT of empty is 0.
DATE	(0-input operation) outputs a sentence of 4 elements: the first is the day of the week, the second is the year, the third the name of the month, and the fourth the day of the month. For example, "MONDAY 1973 OCTOBER 8".
DIFFERENCE DIFF	(2-input operation) given two numeric words, DIFFERENCE outputs the arithmetic difference.
DIVISION	(2-input operation) outputs a sentence whose FIRST is QUOTIENT of its inputs, and whose LAST is REMAINDER of its inputs.
DO	(1-input command) executes the logo statement specified by the input, which is a word or sentence indicating a Logo command. For example, "DO "PRINT 1"" will cause 1 to be printed. An input may be a command, or it may be a command prefixed by a line number: DO "10 PRINT 1" inserts "PRINT 1" as line 10 of the function currently being edited.
EDIT	(command followed by procedure name) causes you to enter editing mode, with the specified function made the currently edited function.
EITHER	(2-input operation) outputs TRUE if either input is TRUE. If both arguments are FALSE, it outputs FALSE.

EMPTYP	(1-input operation) outputs TRUE if its input is EMPTY; otherwise it outputs false.
END	(0-input command) causes an exit from definition or editing mode into immediate execution mode.
EQUALP	(2-input operation) outputs TRUE if its inputs are numerically equal, and FALSE otherwise. Two numbers are compared numerically by disregarding leading zero's, and taking signs into account.
ERASE	(command followed by procedure name) deletes the definition of the specified procedure.
ERASELINE	(1-input command) deletes the line specified by the input from the currently edited function.
FIRST F	(1-input operation) outputs the first element of its input; generates an error if the input is empty.
GET	(command followed by file name) adds the data stored in the file to the current workspace.
GOODBYE BYE	(0-input command) causes Logo to return control to to the operating system.
GREATERP	(2-input operation) outputs TRUE if the first argument is numerically greater than the second, and FALSE otherwise.
GOTOLINE GOTO GO	(1-input command) the input must be a line number in the currently executing function. Control is transferred to the specified line.
IFFALSE IFF	(command followed by statement) if the truth flag is currently set to FALSE, then the statement is executed; otherwise, nothing happens.

IFTRUE IFT — (command followed by statement) if the truth flag is currently set to TRUE, the statement is executed; otherwise, nothing happens.

IGNORE — (1-input command) does nothing at all.

IS — (2-input operation) the inputs may be any Logo things. IS returns TRUE if they are the same object, and FALSE otherwise.

LAST L — (1-input operation) outputs the last element of its input. Generates an error if given empty.

LESSP — (2-input operation) outputs TRUE if the first argument is numerically less than the second, and FALSE otherwise.

LINES — (1-input operation) outputs a sentence of the line numbers of the function named by the input.

LIST — (2-input operation) outputs a two-element list, whose FIRST is the first input, and whose LAST is the second input.

LOGOUT — (0-input command) control is returned to the operating system in in such a way that the computer disconnects the communications line.

MAKE — (2-input command) causes the thing of the first input to be the second input. For example,

MAKE "A" "THE COW ATE THE CHEESE"

causes :A: to be THE COW ATE THE CHEESE.

An introductory form of MAKE is also provided. This form is used as follows:

<u>MAKE</u> NAME: <u>BELLA</u> THING: <u>COOLA</u>

Logo prompts the user for the name and thing when the introductory MAKE is used.

MAXIMUM MAX	(2-input operation) outputs the maximum of the two arguments, compared numerically. | f('minimum min',2-input operation') outputs the minimum of the two arguments, compared numerically.
NOT	(1-input operation) outputs TRUE if its input is FALSE, and FALSE if its input is TRUE.
NUMBERP	(1-input operation) outputs TRUE if its input is a number, and FALSE otherwise.
OUTPUT	(1-input command) causes the currently executing procedure to return with the input as its output.
PRINT P	(1-input command) types its input on the user's console, and then types a carriage return.
PRODUCT PROD	(2-input operation) outputs the arithmetic product of its two inputs, which must be numbers.
QUOTIENT	(2-input operation) outputs the quotient of its two inputs.
RANDNO	(1-input operation) outputs a number chosen at random which is numerically less than the number given as input. 0 is a possible output from RANDNO.
RANDOM	(0-input operation) outputs a digit chosen at random. 0 is a possible output.
REMAINDER	(2-input operation) outputs the remainder of its two inputs.
REQUEST	(0-input operation) reads a line, and returns a thing corresponding to the line. <u>The result of REQUEST is always a list.</u>
RESETCLOCK RESET	(0-input command) resets the internal clock.

SAVE	(command followed by file name) saves the procedures and variables of the current workspace in the specified file.
SEED	(1-input operation) outputs the current value of the seed for the random number generator. If the input is non-zero, the seed will be set to the value given by the input, which must be numeric.
SENTENCE S	(2-input operation) outputs a sentence made from the first input followed by the second input. If either input is a word, it is converted to a one-element sentence.
SENTENCEP	(1-input operation) outputs TRUE if its input is a sentence (or a list), and FALSE otherwise.
SHOW	(command followed by procedure name) prints out the definition of the specified procedure. In editing or definition mode, the procedure name need not be specified, in which case the function currently being edited will be displayed.
SHOWLINE	(1-input command) displays the line of the currently edited function specified by the input.
STOP	(0-input command) causes the currently executing procedure to return.
SUM	(2-input operation) the inputs must be numbers. SUM outputs a word which is the arithmetic sum of the two inputs.
TEST	(1-input command) the input, which must be one of the words TRUE or FALSE, is stored away in the truth flag.

TEXT (2-input operation) outputs a line of the function specified by the first argument. For example:

```
TO GREET :X:
10 PRINT "HELLO"
20 PRINT :X:
END
```

TEXT OF "GREET" 20 will output "20 PRINT :X:". Specifiying line 0 will obtain the title line.

THING (1-input operation) outputs the thing of the word given by the input.

TIME (0-input operation) outputs the time of day as a sentence of "hour minute day".

TITLE (command followed by title line) changes the title of the currently edited function to that specified.

TO (command followed by title line) enters definition mode, editing the function specified in the title line. Generates an error if the function is already defined.

TRACE (command followed by procedure name) causes the specified function to be traced: each time the procedure is called, a message will be printed out to indicate the depth of call, and the arguments. Each time the procedure return returns, a message will be printed out, indicating the output, if any, of the procedure.

TYPE (1-input command) prints out its input on the user's console, but does not type a carriage return afterward.

TYPEIN (1-input command) equivalent to MAKE input REQUEST.

WAIT (1-input command) waits the number of seconds given by the input.

UNTRACE	(command followed by procedure name) turns off tracing for the specified procedure.
WORD W	(2-input operation) outputs a word made from the characters of the first input followed by those of the second input. Generates an error if either input is a sentence or list.
WORDP	(1-input operation) outputs TRUE if its argument is a word, and FALSE otherwise.
ZEROP	(1-input operation) outputs TRUE if its input is numerically equal to 0, and FALSE otherwise.

PART 3 BCPL

{ They have been at a great feast of languages, and stolen the scraps.
-- W. Shakespeare, Love's Labour's Lost }

Although it would be out of place to give a complete description of BCPL here (for full details, the reader is referred to [RICH69]); a summary of the language will perhaps aid in understanding the structure of the system.

BCPL is an Algol-like language designed for compiler writing; its two prime ancestors are CPL [BARR63] and Algol 60 [NAUR63]. The language is designed with two particular goals in mind:

- machine independence, even at the cost of a slight decrease in efficiency, and
- "good programming style", in encouraging modularity and discouraging arbitrary use of GOTO's.

The basic datum of the language is the word, or cell. A word is a string of bits: it is presumed to be the primary addressing unit of the computer. While word length is not specified, it is assumed to be at least 16. In particular, a word must be able to contain a machine address.

Since a word is simply a bit sequence, the object in a word can be of any type: an unsigned integer, a signed integer, a character, a pointer, a floating-point number, or a

bit mask. This means that BCPL does no type checking whatsoever--the onus is on the programmer to ensure that he does not use a statement address when, for example, doing floating point arithmetic. It may seem that this leaves the programmer exposed to all kinds of dangers; however, from my experiences while writing BCLogo, I had very few errors that could have been caught by the type-checking machinery of the average Algol 60 compiler--most of my errors were in trying to use NULL as a valid pointer. I have more to say on this topic in the conclusions.

One feature which makes BCPL machine independent is that successive word addresses are guaranteed to differ by 1. This means that even though word addresses on the 360, for example, differ by 4, a BCPL program which runs on a word oriented machine such as a PDP10 will work on the 360.

As far as good programming style is concerned, there are two particularly pleasant BCPL features: the global vector, and the stack. The global vector is simply an array which is accessible from all modules of the program. The programmer may declare that an identifier X refers to cell n of the global vector. If he does so in all his modules, then he can use X in all his modules. This means that X has become a global variable. The whole arrangement is much like Fortran's COMMON blocks, but much less unreliable--the association with common words is not done by ordering within the declarations, but by explicitly specified numbers. Also, since the global vector is an array, one can index into it. This feature is put to good

use by the primitive functions. Each BCPL object module contains a table indicating those global vector locations which are to be preset with specified routine addresses. Therefore, the modules of primitive functions specify that the primitives are to be loaded into specific global vector locations. Then, at initialisation time, commands are read which in effect say "function FOO, located at global vector entry n, is predefined." This means that the system does not have to be recompiled when each new function is added.

Since BCPL is oriented around a stack, it is very easy to allocate storage dynamically for short-term needs, without tying core up over a longer period. Also, even though one wants, for efficiency reasons, to avoid recursion whenever possible, some procedures (especially input/output) are most naturally programmed recursively. A stack makes this possible. One problem which vitiated the use of the stack to some degree during debugging was an error in the BCPL run-time support package which made the system expire noisily when the stack overflowed.

During the planning stages of the thesis, I evaluated a number of different languages. I rejected some, such as Assembler and PL360 as being too machine dependent; also, too much is asked of the person who simply wishes to add a new primitive function.

Fortran can be made machine-independent (although nobody ever really implements ANSI standard Fortran); however, I did not relish the thought of programming in as unstructured a

language as Fortran--in addition, Fortran lacks some of the absolute necessities of a system programming language: dynamic storage allocation, named constants, and bit manipulation.

Algol W and PL/I are also quite reasonable candidates--both have been implemented on a number of machines (one could quite reasonably translate an Algol W program into SAIL [swin73], for example); yet the implementations available on our system did not generate efficient enough code. Of the languages available on our system, only BCPL had the right combination of structure, efficiency, machine independence, and power. I feel that it would have been no harder to have written the system in PL/I or Algol; however, these languages were not economically reasonable on our system.

The results have been quite pleasing: the language is at least as convenient as, say Algol W; yet, of course, the implementation is more efficient than Algol W. Probably the main hurdle was in debugging--since there is currently no symbolic debugging system for BCPL, the main debugging method was the insertion of more output statements. Luckily, the compiler is cheap enough to run that this was no great drawback. Nevertheless, there were a number of times at which I longed for the CHECK conditions of PL/I, or the formatted post-mortem dumps of Algol W. On the whole, however, it was very easy to locate errors--there were almost no errors which were not at once locatable in a given module. BCPL code is at least as readable as that of Algol W or PL/I, and thus there were very few occasions when I asked myself "What was I

thinking when I wrote that?". One significant plus which some PL/I implementations (but not ours) share with BCPL is that procedure addresses are data--therefore, one can have an array of procedures. I used that feature in the administration of procedures. A language which does not allow arrays of procedures would require that primitives be reached by something like a case statement in Algol, or a computed go to in Fortran. This in turn makes new primitive functions harder to add.

PART 4 SYSTEM DESCRIPTION

{ My name is Ozymandias, king of kings
Look upon my works, ye mighty, and despair
-- P.B. Shelley, Ozymandias }

This section attempts to give the reader a feeling for how the system is constructed. For full details, the reader is referred to a forthcoming document [MANI73].

BCLogo is quite modular, in that the components of the system are relatively independent. Almost all communication between components occurs either in global variables or in the 'data base'--the garbage collected area. The components presently in the system are:

- the parser is used to convert from character strings into internal program format. Later, when infix operators are added, the parser will be extended to put the operators in the right place.
- the interpreter is used to execute program lines. It scans each line, and evaluates all the function calls of a line.
- the storage manager and garbage collector allocate space. When it is not possible to allocate more space, the garbage collector then frees all 'useless' space--that which is not currently being referenced. When it is not possible to reclaim any space at all, the system simply retires with the comment "Logo needs more space--see your teacher". Minor

modifications could be made to allow the system to acquire more store dynamically at this point.

- the primitive functions accomplish the actions specified by the Logo built-in functions.
- the initialisation routine clears the workspace, and puts in the definitions of the built-in functions.
- the unparser converts from internal program format to character string format.
- a number of miscellaneous routines handle errors and interfacing to the system.

4.1 Data Structures

BCLogo uses two primary types of data: fixed length blocks, or nodes, and variable length strings, or print names. Any Lisp programmer will feel at home with the internal structure:

- words are stored uniquely using an Oblist.
- sentences are linked lists of nodes.

A word is represented internally by a data structure called an atom, which is composed of two parts: the atomhead is a node containing a pointer to the Thing of the word, a pointer to the user procedure (if any) defined for this word, switches such as whether the procedure is traced or not, and a pointer to the atom tail, which contains the print name representing the word. The structure of an atom is shown in figure 4-1.

```
MAKE "Q" "W"
TO Q
10 PRINT "HELLO"
END
```

Oblink	Chain pointer for atoms in same equiv. class
Thing	Pointer to atom "W"
Proc	Pointer for control block for Q procedure
Pname	Address of the char string "Q"
Switches	Flags, e.g. number of arguments, trace flag.

FIGURE 4-1

THE FORMAT OF AN ATOM

Since atoms are represented uniquely, we must have some means of determining for a given word what atom represents it. I have followed the standard Lisp solution to this problem: the Oblist, which is simply an array into which addresses are computed by means of a hashing function on the print name. Each element of the array points to a linked list of atoms, each of which is in the same equivalence class. To find a given word, the system searches along the particular equivalence class. If the word is found, the address of the existing atom is used for the word. Otherwise, a new atom is constructed and added to the Oblist; its address is then used.

This process sounds quite expensive. However, it is only necessary to intern (using Lisp jargon) a word when it is created, either by a primitive function, or by reading it in.

As an example of the use of atoms, suppose that we had executed

```
MAKE "GHOTI" "FISH"
TO GHOTI
10 PRINT "PSHAW"
END
```

The resulting data structure is shown in figure 4-2.

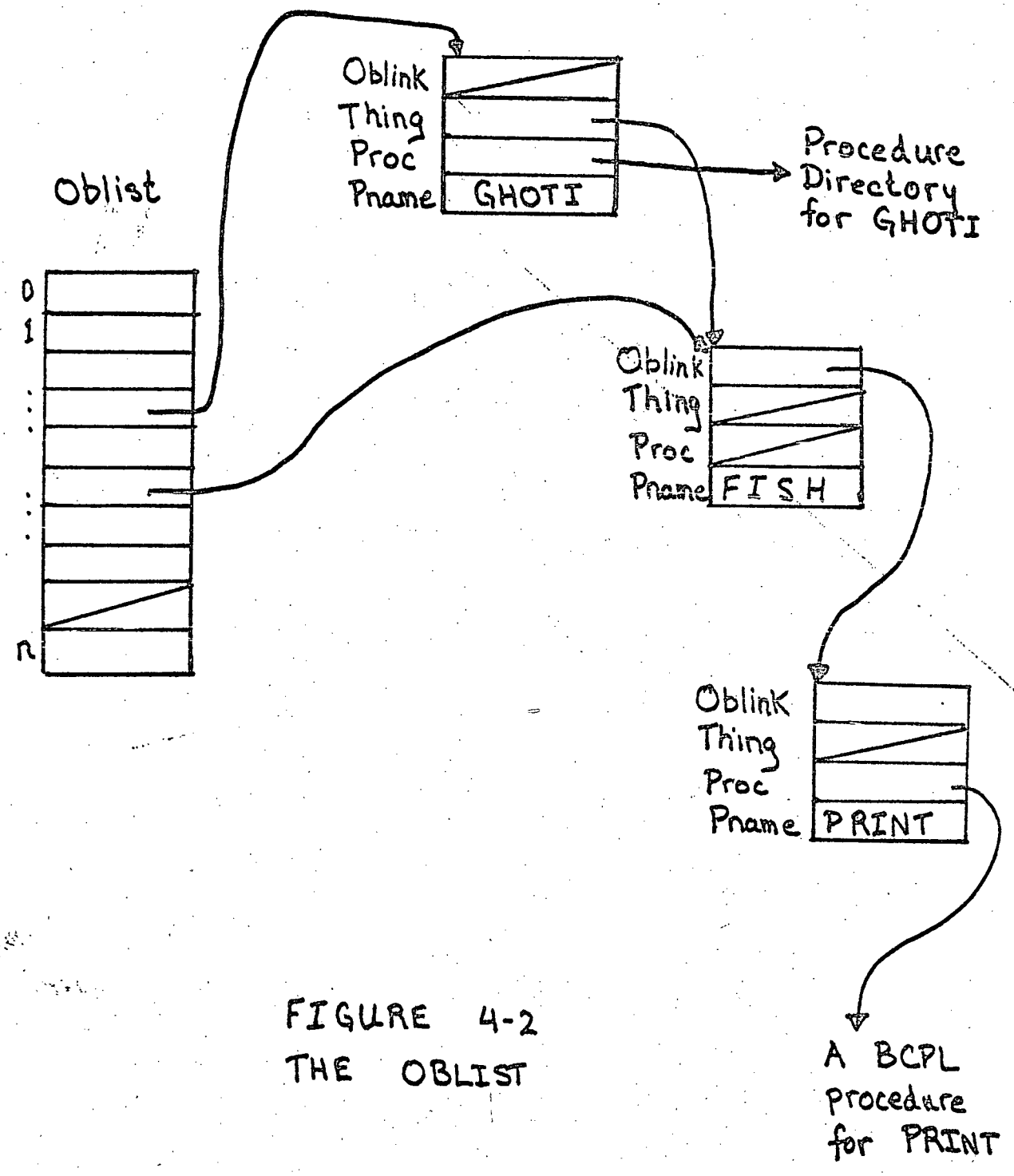


FIGURE 4-2
THE OBLIST

4.2 Storage Management

{ All the rivers run into the sea; yet
the sea is not full.
-- Ecclesiastes }

The storage management problems of a list processing system such as Logo are probably the most important ones. I attempted to build a storage management system which would let me forget about these problems when implementing the rest of the system. I have had somewhat mixed results. The resulting system was so convenient to use that I went about blithely creating nodes without any care as to whether the garbage collector could handle (or even know about) them.

The basic strategy which I followed was constrained by the following factors:

- since I wanted to be able to put the system onto a small computer, I could not afford a 'copy' style of garbage collection such as that of [HANS69].
- I did not wish to worry about paging problems, since the average Logo data structure is very small.

The resulting storage management system has the following characteristics:

- node storage (the "heap") is divided up into pages; all the pages are of the same size (for any reasonable efficiency, the pagesize must be a power of 2).
- the programmer defines each type of node he is going to

use to the storage manager. Each type of node is of fixed size. The words of a node may contain either pointers to other nodes (references) or other data. All nodes of a given type contain the same number of references--in the node, all the references precede the other data.

- all of the nodes on a given page are of the same type.

In order to emphasize the independence of the storage management package from the remainder of the system, the following paragraphs pretend that the storage manager is of general use, and outline how the package is used.

The programmer defines each of his node types by a calling a procedure with a unique integer code which identifies the particular type, and two lengths: the length of the node, and the number of references. This procedure initialises certain table entries.

At any time after this initialisation call, the programmer may create a new node of a specified type by calling the procedure 'new'. This procedure is given a type code: it does anything necessary (including if necessary a garbage collection) to allocate a specified node.

In general, the programmer is not encouraged to free nodes. There is a procedure provided to do so; however, the programmer may destroy a node to which another node still points. The recommended way to reclaim data is by using the garbage collector. This routine follows all pointers into the node area, marking all nodes which can be reached from one of

these pointers. It then proceeds to free all unmarked nodes, since these unmarked nodes cannot currently be used. All freed nodes are placed onto a "free list", of which there is one per page. Therefore, all that 'new' has to do is to go to a page containing the right kind of nodes, and remove one.

Print names are collected in a different way: since they are variable in length, one cannot simply use a free list approach--fragmentation would result. Instead, each print name has a back link to its atomhead. Then one can linearly scan the print names--for each print name, one checks to see whether its atomhead is marked. If so, one simply moves the string to the valid string area, modifying the atomhead appropriately. Otherwise, the string can simply be ignored. This approach is taken from [FEUR71].

There is one last refinement we can make: there is a large class of words which appear for a while, but are then useless. For example, if we read in the atom "FOO", operate upon it for a while, and then forget about it, we have FOO forever, since it is pointed to by the Oblist. However, if we have forgotten about it, we might as well reclaim its storage. Therefore, during each collection, the garbage collector examines each atom; if an atom is "truly worthless", in that it is neither marked, nor the name of some Thing, nor the name of a function, it is deleted.

There is nothing very startling about this whole design--it seems to have been the way the original Lisp implementation was done. The reasons I chose it are that it is very simple to

program, and that the allocation of a node is relatively fast, since in most cases all that needs to be done is to pop a node off a free list. The page idea came from two directions: the Algol W system [BAUE68] uses a very similar design; also, a conversation with Wallace Feurzeig and George Lukas convinced me that everything should be in the heap. This meant that there were to be a relatively large number (currently 7) of different node types, which meant that either one had to have 7 different core areas, or that the garbage collection problem would become messy. The page solution solves this problem, allowing quite a wide variations in the amount of space for each node type. Yet another reason for dividing storage into pages came from the facility that some operating systems have for allocating store dynamically. While the current storage manager assumes that the pages are contiguous in store, only a small number of changes need to be made to allow pages to be allocated dynamically. This feature allows much more flexibility than the use of separate arrays for the various types of nodes. See figure 4-3 for the arrangement of page tables.

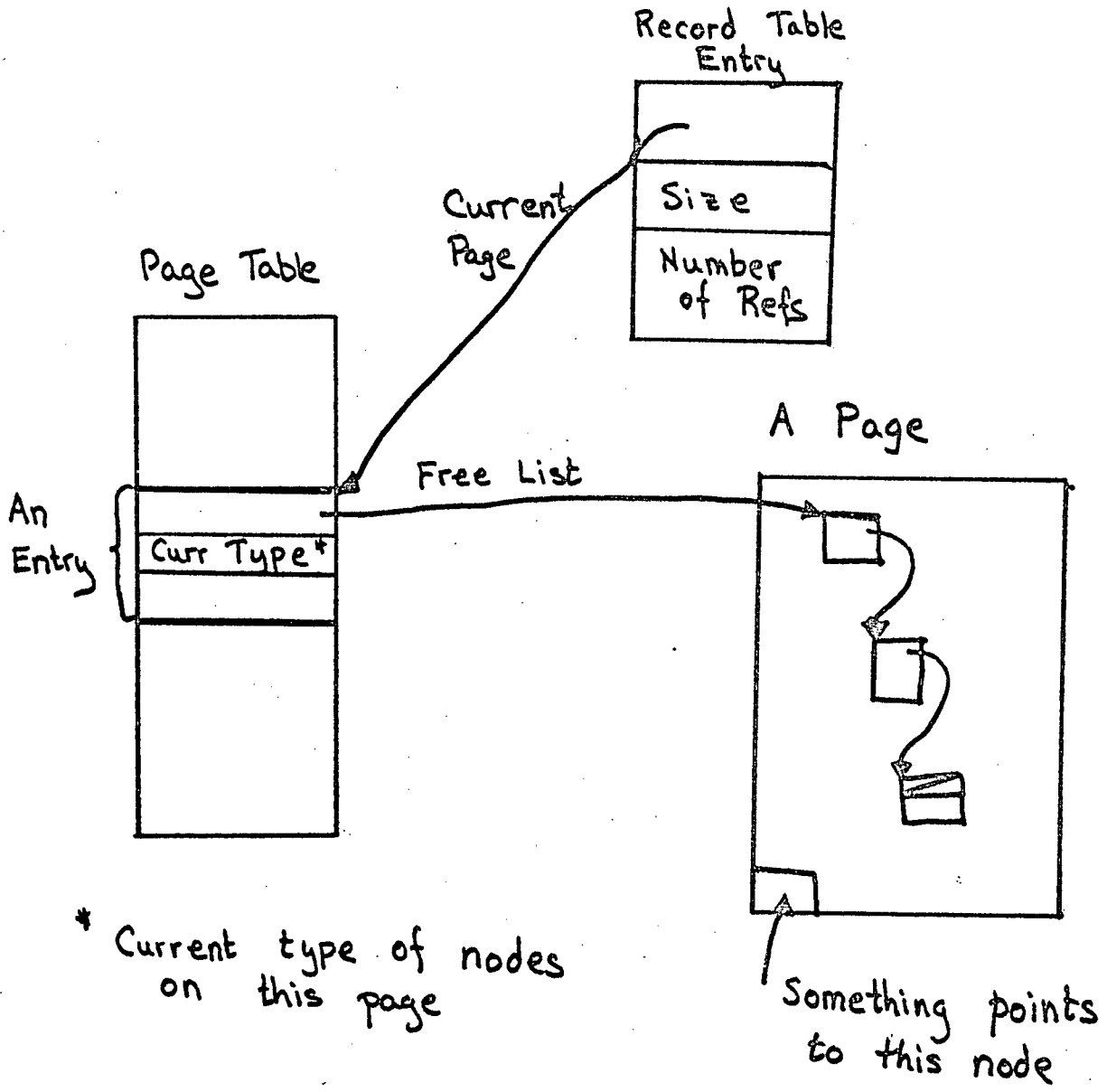


FIGURE 4-3

PAGE FORMAT

4.3 The Parser

{ And I copied all the letters in a big round hand!
-- W.S. Gilbert, H.M.S. Pinafore }

There really isn't much to say about the parser. It operates in two passes: the first is essentially a lexical scanner, which builds list structure from the string of characters which the user enters. The second pass goes over the list structure, checking that noise words and parentheses are correctly used. This pass will later be extended to handle infix operators.

4.4 The Interpreter

{ It revolts me, but I do it!
-- W.S. Gilbert, The Mikado }

The basic objects with which the interpreter itself operates are the line being interpreted, and the push-down stack. The stack contains a chunk of data (a frame) for every pending function call. Each frame contains the following information:

- the name of the function to be called.
- the arguments of the function.
- various global variables which must be saved, such as the position within the current line, the line being executed,

and the truth flag.

Since the interpreter scans a line from left to right, it sees the function name before it sees the arguments. Therefore, on seeing a new function name, the interpreter builds a new frame for the function. As each argument is scanned, it is added to the frame. Finally, when all the inputs have been evaluated (the interpreter, of course, knows how many inputs each function is expecting), the function under consideration is called. If it is a primitive, it is called right away; if it is a user procedure, the interpreter stores its global variables away in the new frame, and changes them to make it appear that it is now executing the called function.

At this time, let's look at an example:

```
TO DOBLE :X:
10 OUTPUT SUM :X: AND :X:
END
PRINT LAST OF DOBLE OF 34
```

The following sequence happens:

- we start off by encountering PRINT. Since this is a function name, we build a new stack frame.
- next, we encounter LAST. We build a new frame for it.
- OF is a noise word--we ignore it.
- DOBLE is a function name; therefore, we build yet another frame.
- 34 is a datum. We add it to the current stack frame (the arguments for DOBLE).
- now, seeing that DOBLE takes one input, we call it, by changing our attention to the definition of DOBLE. But first, we set X to 34.
- we start executing the first line of the function: line 10.
- OUTPUT is a function, so we create another stack frame for it.
- SUM causes another stack frame to be created.
- :X: is a variable; therefore, we get its Thing, 34, and

place it on the stack.

- we see that we have not yet got enough inputs for SUM. Therefore, we do not call anything.
- we ignore the noise word AND.
- we evaluate :X: again, putting its value onto the stack.
- now we have all of SUM's inputs--we can call SUM. It returns right away, with a new value-68. We put this value onto the stack.
- now we notice that we have satisfied OUTPUT's requirements for inputs. We call it; it immediately forces a return from DOBLE. Now we take OUTPUT's input, and place it on top of the current stack frame, which is the frame for LAST.
- now we have satisfied LAST. We call it, receiving immediately its output-8.
- PRINT has now received its input. We call it with 8; it prints out 8, and returns,
- there is nothing more to do, so the interpreter returns to get the next command.

The interpretation algorithm seems to have come from thin air; yet a little thought should convince the reader that it is exactly the algorithm which one would use when explaining how Logo works. The only difference is that one would normally say "and when you call a user defined function, the interpreter calls itself". However, it is obvious that a recursive implementation of the interpreter would consume a great deal of storage. Even worse, if the user were to execute a procedure such as

```
TO GROAN
10 GROAN
END
```

The first thing to happen would be that the interpreter would go into an infinite recursion loop. Eventually the BCPL stack (which is used to store bookkeeping information about each BCPL function call) would overflow, and, since BCPL doesn't check for this unfortunate occurrence, a program fault would occur. Therefore, we must make our interpreter iterate--a non-trivial task for something which is so easy to do recursively.

4.5 Variable Binding

So far, we have glossed over what happens to the names of the inputs of a procedure. Obviously, one must save the previous values of the input names: for example, in our DOBLE example, we may have had an X somewhere else; we would be greatly distressed if that X were overwritten during a call to DOBLE. Therefore, when it calls a procedure BCLogo saves the previous value of the variable on the stack, and sets the Thing of the variable to the specified input. When the procedure returns, the whole process is reversed--X is restored to its initial value. This type of process is called "shallow binding". The point is that the value of a variable V is always the Thing of V. One never looks on the stack for the value of the variable--it is always in the Thing cell. While this scheme is the most obvious--it is used in several Lisp implementations (for example, MACLisp [WHIT70] and Lisp/MTS [WILC73])--it is not the most general. Consider for example a world in which we had two processes, each with its own private V--for example, two independent robots each fighting its way out of a labyrinth. Since each robot has his own value of V, we can't very well use the Thing cell to store the values. The generally accepted way of handling this sort of situation is to use a tree of stacks [BOBR72], always storing the value of a variable on the stack. This method, called "deep binding" is far more general, at the cost of a small loss in efficiency. I chose the shallow binding method not because of its small gain in efficiency, but because Logo at present does not need deep

binding. If we ever want to allow parallelism or backtracking, only small changes to BCLogo will be needed in order to implement deep binding.

4.6 The Primitive Functions

Since one of the main goals of this project has been to construct an easily extendable system, a great deal of effort has been spent in making the writing of primitive functions very easy. In particular, it is not necessary to recompile the system simply to add a new function. All that is necessary is compilation of the function (along with a gigantic file of declarations), loading (or link-editing) the object code for the function with the rest of the system, and then making the appropriate entries in the initialisation file. This means, in particular, that new types of graphics support, or file management, or anything else, may be added at any point. All that is necessary is that the function programmer have a working knowledge of BCPL, plus a minimal understanding of the system.

As a test of the ease of adding new functions, I spent about ten minutes explaining the method of adding new functions to another student in our department, Peter VandenBosch, who then wrote some turtle functions. He had no problem adding these functions to the system.

The rules that a primitive function must follow are remarkably simple:

- all primitive functions are BCPL functions (i.e., they always return a value). They always receive one input, which is set to the number of inputs specified by the user in this call. (Since most user functions will take a fixed number of arguments, in general this parameter may be ignored).
- the inputs to the function are in the array ARG. The first is in ARG!1, the second in ARG!2, and so on.
- when the function has finished its operation, it always returns something; if it is an operation, it returns the resulting value. If it is a command, it returns the constant NOTHING.
- a number of useful routines are provided for the programmer's use:

INTERN(BUFF)

this procedure takes a BCPL unpacked string (one in which word 0 contains the number of characters, and the remaining words contain the characters). It returns a pointer to the atomhead for the Logo version of the string.

ITOS(NUM)

this function takes a binary integer, and returns a pointer to a Logo atom corresponding to the integer.

STOI(ATOM)

given a Logo atom, this procedure returns the binary representation of the input. If the atom contains non-numeric characters, it will generate an error.

CONS(A,B)

given two Logo pointers, this function will return a pointer to a new cell, Car of which is A, and Cdr of which is B.

ERROR(A,X,Y,Z,W,...)

this super amazing function prints error messages. A is an integer code indicating which error message from the initialisation file is to be displayed. X, Y, and the other parameters are data which are inserted into the error message. These parameters may be any BCPL data; they may also be pointers to atoms or cells. Please see the description of the initialisation file for more details.

As an example, a listing of the primitive for 'WORD' follows.

```
GLOBAL $( WORD: FT+71 $)
LET WORD(I) = VALOF
$( LET V = VEC 255 // RESULTS ACCUMULATED HERE.
   AND U = VEC 255 // TEMPORARY.
   AND L = 0 // # CHARS ACCUMULATED SO FAR.

   FOR J = 1 TO I DO $(
      // CHECK EACH ARGUMENT.
      LET ARG = ARGS!J
      IF ARG = NULL THEN LOOP
      IF TYPE(ARG) =ATOMHEAD THEN ERROR(50, ARG)
      UNPACK(ARG, U)
      FOR K = 1 TO U!0 DO
      $(
         L := L+1
         V!L := U!K
      $)
   $)
   V!0 := L
   RESULTIS INTERN(V)
$)
```

More examples may be found in a forthcoming document on turtle geometry [MANI73B].

4.7 Initialisation

BCLogo is initialised by reading in a file of commands. These commands cause procedures to be defined, error messages to be stored, initialisation procedures to be called, and messages to be typed to the user, as well as setting various options, such as whether a dribble file (a file in which a record of the complete dialogue is stored, for later perusal by a teacher) is to be kept, the amount of space to be allocated, and other parameters.

This approach was chosen for two reasons: first, it means that only the initialisation routine need stay in memory

throughout execution; therefore, the data used for initialisation disappear right after they are used. Naturally, there is a disadvantage: there is an extra file opening and closing operation, plus a number of input operations, for the initialisation file. This means that the system takes a bit longer to load than it would if the initialisation data were built right into the system. However, on the 360/67, only about a second of CPU time is involved--an amount I hardly feel worth worrying about.

The initialisation data is in the form of lines, each of which starts with a command letter, followed by data for that command. The available commands are:

- C (call) this command takes one parameter, the index into the global vector of a procedure which is to be called. This command lets packages of functions (such as the turtle functions) provide an initialisation entry point.

- D (define) This command takes four, count them, four parameters. The first of these is the name of a function to be defined. The second is an index into the global vector for the procedure to be called. Next comes the number of arguments for the function; a variadic function such as SENTENCES is flagged by coding 99 here. The last parameter is usually 0; it indicates whether the function is 'normal' (in the sense mentioned when discussing the interpreter). For an 'abnormal' function, code a 1.

- E (error) defines an error message. It takes two parameters: the first is the number of the error message, and the second is the text to be printed out when that error

message is produced. The text may have embedded editing codes, flagged by a preceding percent (%) sign, as follows:

% print a percent sign.

C print the next argument as a character.

In print the next argument as an integer in a field of width n.

N print the next argument as an integer, in minimum width.

P print the next argument as a Logo datum (Atom or Cell).

S print the argument as a BCPL string.

- O (option) sets the various option flags from the remainder of the line, as specified by characters in the line. The following characters currently do something

D enables dribble file.

N indicates that you are creating a "new system", and want some additional reassuring information during initialisation. You will then get a summary of how the workspace was subdivided, and a few extra timing results.

G causes a message during each garbage collection; only of interest to systems freaks.

- X (exit) causes initialisation to terminate, and regular processing to begin.

This method of initialisation (by reading commands from a file) is quite convenient, although, of course, hardly new--programs have been reading in initialisation data for years.

The exact form of the input comes from thoughts on how initialisation ought to be done--thoughts inspired by the manner in which syntax-directed compilers read in the descriptions of the languages which they will compile today.

PART 5 MACHINE INDEPENDENCE

5.1 BCPL

As one of the prime aims of the project was the production of a machine-independent system, it is necessary to evaluate how well BCPL served this goal.

BCPL, although a sem programming language[1]yst isolates the programmer from the machine and the host operating system; the only exception is that a library of machine-dependent procedures is provided for input/output and other systems tasks. The only machine dependent features of the language itself are those of data format and word size, where, obviously, a program depending upon the two's complement arithmetic of the 360 will fail utterly when run on a one's complement machine such as the Univac 1108.

In order to comply with the least common denominator, I have pretended that a word was 16 bits wide. While a few masks assume a 32 bit word, there is nowhere an attempt to squeeze more than 16 bits worth of information into one word. (Addresses appear to be an exception, but on almost all computers,an address fits into one word).

---

[1] at least one operating system has been written in it [STOY72].

The only exception to this rule is in the garbage collector, which assumes that a word can hold an address plus one extra bit, which is used during the marking phase of garbage collection. Since some minicomputers need every bit of a word to store an address (for example, the PDP-11), one may have to rewrite the garbage collector to make use of a bit map, which contains one bit for every word of the workspace.

In general, achieving machine independence means losing efficincy). Three separate inefficiencies show up:

- since BCPL runs on a hypothetical machine in which successive word addresses differ by 1, a BCPL address is a word address. On the other hand, a 360 machine address is a byte address. This means that every time we follow a pointer, we must convert from one to the other. Even though the BCPL compiler employs an optimisation which cuts down the excess overhead to two bytes of storage and less than one microsecond, it still adds up.
- the most efficient way to store strings is to pack them four (on the 360) to a word. Yet the only machine-independent way to process a string is to store it one character per word. This means a great deal of packing and unpacking, which is totally unreasonable. One reasonable optimisation is to provide an assembler version of INTERN, the routine which does most of the character string handling--the parser could also be rewritten in assembler, but it is not clear that the savings would be very great from such a rewriting.

5.2 Putting BClogo On Another Machine

If BCLogo is truly machine independent, it should be very easy to move from machine to machine. We shall consider two separate versions of the problem--moving from operating system to operating system, and then moving from machine to machine. The following section considers the problem in the abstract. For full details, the reader is referred to [MANI73].

5.2.1 Operating Systems

Changing operating systems should be very easy. There are only a few places in the system which depend upon a particular operating system. The only dependencies are those concerning files: the routines OPENINPUT, OPENOUTPUT, and CLOSE within the main segment of code may have to be rewritten for each operating system. Apart from this, the only real problem is that many operating systems do not allow enough control over the such things as attention interrupts. If it is desired to run BCLogo under such an operating system, it will not be possible for Logo to catch attention interrupts.

Therefore, transporting BCLogo from one operating system to another on the same machine means rewriting only a small amount of BCPL code, plus modification to the BCPL system to run on the new system.

Note that it is not possible to run BCLogo on an

operating system which makes unwarranted demands, such as requiring that certain registers always contain addresses, or that a certain register always point to some area. This means, for example, that BCLogo is not operable under CALL/360.

5.2.2 Changing Machines

The greatest amount of work in moving from machine to machine is converting BCPL to run on the new machine. This means that the codegenerator of the compiler (less than 1000 lines of code) must be rewritten to generate object code for the desired machine. At this point, all that is necessary is recompilation of the BCLogo system, thus generating object code for the new machine.

PART 6 CONCLUSIONS

6.1 Rough Spots

Although a very reasonable system is currently running, it should be obvious that BCLogo is nowhere near completion. There are a number of rough spots--in particular, some of the error checking is not quite foolproof.

Nonetheless, I feel that the basic framework is now present. The system is modular enough that a large number of incremental changes can be made without significantly changing the basic structure. Specifically, in the near future, I intend to rewrite both the storage manager and the interpreter, in order to correct the errors mentioned above, and to change some of the data structures--notably the stack. These changes should be completely transparent to the rest of the system.

There is one significant problem for which I currently have no solution. In order to be machine independent, we must use a language such as BCPL, which deliberately sets out to isolate the user from the machine. Yet this isolation costs a great deal, both in the efficency degradation mentioned in the previous part, and in the overhead of wasted procedure calls. For example, if the Logo programmer issues a turtle command,

the following sequence of calls is necessary: the Logo procedure calls the BCPL turtle primitive, which calls an interface module in Assembler, which calls a Fortran routine, which calls the system plot routines. This degree of nesting seems to me to be a bit excessive.

Regarding the use of pointers, I feel that my system suffers from a problem in its design for garbage collection. The problem is that since garbage collection can occur at any time, it is neccessary to ensure that all pointers into the heap are accessible to the garbage collector. But consider a recursive list building routine: it needs a number of pointers to the list that it is manufacturing. Naturally, the garbage collector must know about these pointers. Yet, since the routine is recursive, it is not adequate simply to make the pointers global, since there is one set of pointers for each level of the routine. The only reasonable solution to this problem is to put the pointers onto the stack. My present system does not do this.

6.2 Friendliness

So far, I have not been able to do anything about making the system "friendly". There are a number of hypotheses one can frame about the friendliness of a system, such as:

- a system is friendly if it can be anthropomorphised by the user, giving him the illusion that he is conducting a conversation with a human being. Systems with this property are Eliza [WEIZ66] (for 5 minutes or so), and the INTERLISP programmer's assistant [TEIT71] (to someone whose native language is Lisp). This may be a very dangerous illusion to promote.
- a system is friendly if it behaves impersonally, thus not giving the user the feeling of a harsh critic who comments on each command (very often adversely).
- a system is friendly if it does what the user means, not what he says [TEIT71].
- a system is friendly if, when it doesn't understand something, it does not presume on the relationship it has built up with the user, contenting itself with commenting on the fault(s) it has observed.
- a system is friendly if it immediately reports errors it has found.
- a system is friendly if it waits to report errors until the last possible moment, commenting upon as many errors in the command or program as it can.
- a system is friendly if it gives only terse messages, thus not frustrating the user by making him wait for a long

message, the contents of which he already knows.

- a system is friendly if it explains each error thoroughly, so as to correct as many wrong ideas, diplomatically, as possible.
- [SEEL73] a system is friendly if it promotes the "growth" of the user (it encourages learning about the user's programming environment and about himself). Put another way, a system is friendly if it cares about the user.

The astute, or at least awake, reader will have noticed some contradictory tendencies among the hypotheses. I would venture that the perceived personality of a computer program varies from user to user, and that no one system will be friendly to everyone. Nevertheless, I hope that we can use BCLogo to test out these hypotheses, and others, in the near future.

6.3 Future Developments

In the near future, I hope to add a number of refinements to the system, including floating point arithmetic and infix operators. Neither of these features is of a great deal of worth, since floating point arithmetic may easily be implemented in Logo, storing a rational number as a sentence, either of numerator and denominator, or of mantissa and exponent. However, rational numbers add a great deal of appeal to the language for those who like languages such as Basic.

Infix operators have about the same utility. The only

reason one puts them in is to conform to our preconceptions about mathematical notation. Yet it is not clear that cluttering the language with infix arithmetic does anything. Perhaps it is worth while encouraging children, especially, to write 'SUM 1 2' rather than '1+2'; at least the former notation makes clear that SUM is a function. Nevertheless, one must eventually bow to historical conventions

Our other plans are to manufacture a number of BCLogo's for differing mini-computers. The hope is that Logo running identically on differing computers will be more competitive with the plethora of BASIC's currently in existence.

Bibliography

[BARR63] Barron, D.W., Et Al., "The Main Features Of CPL." Computer J., 6(1963) pp134-143.

[BAUE68] Bauer, H, Algol W Implementation. Department Of Computer Science, Stanford University, 1968.

[BOBR72] Bobrow, D.G., "Requirements For Advanced Programming Systems For List Processing." Communications Of The ACM 15(1972) pp618-627.

[DIJK72] Dijkstra, E.W., "The Humble Programmer". Communications Of The ACM 15(1972) pp859-866.

[FREE73] Freeman, P, "Functional Program Testing And Machine Aids." Program Test Methods, Prentice-Hall, 1973; Pp49-56.

[FENI69] Fenichel, R.R., "A Lisp Garbage Collector For Virtual Memory Computer Systems." Communications Of The ACM 12(1969) pp611-612.

[FEUR71] Feurzeig, W., Et Al., Programming-Languages As A Conceptual Framework For Teaching Mathematics. Bolt, Beranek, And Newman, Inc, 1971. Technical Report 2165.

[HANS69] Hansen, W.J., "Compact List Representation." Communications Of The ACM 12(1969) pp499-507.

[LUKA72] Lukas, G., "Use Of The Logo Programming Language In Undergraduate Instruction." Proceedings Of The 1972 Annual Conference Of The Association For Computing Machinery, pp1130-1135.

[MANI73A] Manis, V.S., "BCLogo Internal Specifications." Department Of Computer Science, University Of B.C.

[MANI73B] Manis, V.S., "Turtle Geometry In BCLogo." Department Of Computer Science, University Of B.C.

[NAUR63] Naur, P, Ed., "Revised Report On The Algorithmic Language Algol 60." Communications Of The ACM 6(1963) pp1-17.

[PAPE71] Papert, S., and C. Solomon, "Twenty Things To Do With A Computer." MIT Artificial Intelligence Laboratory, Logo Memo 3, 1971.

[PAPE72] Papert, S., "Making A Theorem For A Child." Proceedings Of The 1972 Annual Conference Of The Association For Computing Machinery, pp345-349.

[RICH69] Richards, M., BCPL Reference Manual. University Mathematical Laboratory, Cambridge University, 1971.

[SEEL73] Seeley, D.A.R., Private Communication.

[STOY72] Stoy, J.E., and C. Strachey, OS6--An Experimental Operating System For Small Computers." Computer Journal 15(1972) pp117-124, 195-203.

[SWIN73] Swinheart, D., and R. Sproull, SAIL Reference Manual. Stanford Artificial Intelligence Laboratory, 1973.

[TEIT72] Teitelman, W, "Automated Programming--The Programmer's Assistant." Proceedings Of The 1972 Fall Joint Computer Conference, Pp917-921.

[WEIN71] Weinberg, G.M., PL/I: A Manual Of Style. McGraw-Hill, 1971.

[WEIZ66] Weizenbaum, J., "ELIZA--A Computer Program For The Study Of Natural Language Communication Between Man And Machine." Communications Of The ACM 9(1966) pp36-45.

[WHIT70] White, J.L., An Interim Lisp User's Guide. MIT Artificial Intelligence Laboratory, Report 190, 1970.

[WILC73] Wilcox, B., and C. Hafner, Lisp/MTS User's Guide. University Of Michigan, Mental Health Research Institute, 1973.

{ Better is the end of a thing than the beginning thereof.

-- Ecclesiastes }