ELIZA SCRIPTWRITER'S MANUAL

A Manual for the Use of
the ELIZA Conversational Computer System

Paul R. Hayward

Education Research Center

Massachusetts Institute of Technology

Cambridge, Massachusetts

1968

ACKNOWLEDGEMENTS

The writing of this manual was made possible through the funds and facilities of the Education Research Center. I thank Dr. Edwin Taylor for his ideas, guidance, and help in editing the manual. Dr. Walter Daniels has been very helpful with the programming aspects of the system, especially while I was learning the system. Michael Knudsen wrote Chapter 9 on The Slide-Dictionary System. Finally, I thank my wife Bonnie, who helped with the typing and editing, and who has contributed most of all with her patience, understanding, and love for me.

Paul R. Hayward
February, 1968

INTRODUCTION

The ELIZA program provides a method for carrying on teletyped conversations in a natural language between students and a computer. ELIZA is named for Eliza Doolittle, the central character of Pygmalion and My Fair Lady, who, we are sure, can be taught to speak better than she does now. The program was developed by Professor Joseph Weizenbaum of the M.I.T. Department of Electrical Engineering.(1,2) The M.I.T. Education Research Center, together with collaborators around the country, is beginning to apply the program to tutoring students and assisting them in calculations and problem-solving. Some of the present capabilities of the system are described in a paper by Dr. Edwin Taylor of the M.I.T. Department of Physics and the Education Research Center.(3) The use of small discussion units with the program, which is described in the manual, was developed in my bachelor's thesis for the M.I.T. Department of Physics.(4) The computer presently being used with the ELIZA program is the large, general purpose, IBM 7094 in time-sharing mode at the M.I.T. Project MAC and at the M.I.T. Computation Center.

This manual is intended for both inexperienced users learning how to write scripts and experienced users checking on some details of the system. Thus, some sections may seem especially incomprehensible to the beginner. I suggest that the beginner read Chapters 1 and 2, and then try to write and play some scripts. The rest of the chapters contain information not directly related to this first step of writing a script. For the experienced scriptwriter, the flow chart of the system on page A30 may be useful in gaining a better understanding of how the system works. I must add a word of warning to all. The ELIZA system is constantly being changed and improved, so that within a short time there will probably be sections of the manual that will be incomplete or incorrect. The Education Research Center will be able to supply information regarding any changes.

---

1. Joseph Weizenbaum, "ELIZA - A Computer Program for the Study of Natural Language Communication Between Man and Machine" in Communications of the ACM, Volume 9, Number 1, page 36, January 1966.
2. Joseph Weizenbaum, "Contextual Understanding by Computers" in Communications of the ACM, Volume 10, Number 8, page 474, August 1967.
3. Edwin F. Taylor, "The ELIZA Program: Conversational Tutorial" in 1967 IEEE International Convention Record, Part 10.
4. Paul R. Hayward, Flexible Discussion Under Student Control in the ELIZA Computer Program, Bachelor's Thesis, M.I.T. Department of Physics, June 1967.

Chapter 1 -- THE BASIC ELEMENTS

The basic problem in natural language conversation is how to make the computer understand, as well as possible, what the student is saying. In the ELIZA program, this is done by searching for certain "key" words and patterns in the input--the student's part of the dialogue. An important property of ELIZA is the fact that the rules for analyzing and responding to the input sentences are not part of the ELIZA program itself, but are treated as data. They form what is called a script, and are written in a specific format. This means that ELIZA is not restricted to a certain set of recognition patterns and responses, or even to one natural language, for these are specified by the scripts and not by the program.

A. The Script

A script is divided into two sections. The keyword section contains the keywords, along with decomposition rules and reassembly rules (defined below), which are used in the analysis of the input. The program section contains labels and commands or functions that instruct the computer what to do when a student input is (or is not) "recognized". These include printing, calculating, and storing information. Scriptwriting is the means of instructing the computer: "When he says this, you say that."

B. An Example Script

WOOD SCRIPT is a simple script that illustrates some of the basic properties of a script.

```
    WOOD SCRIPT

(CHAIR      (10 CHR   (
            (0 NOT 1 CHAIR 0)  (IT IS A 4 .) XX
            (0 CHAIR 0)  (IT IS A 2 .) YY
            ) NOKEY ))
(TABLE      (TBL      (
            (0) (IT IS NOT A TABLE.) XX
            ) NOKEY ))
(PROG       (WOOD
            GOTO(POPTOP(DAHIN)).
*START      TYPE('I AM THINKING OF AN OBJECT MADE OF WOOD
              FOUND IN A HOUSE.  WHAT IS IT.QQ ').
*XX         TYPE('YOU ARE WRONG. ', SEMBLY).
*YY         TYPE('YOU ARE RIGHT. ', SEMBLY).
*NOKEY      TYPE('NO. TRY AGAIN. ').
            END)
```

A conversation might go like this:

```
r eliza
W 1051.5
WHICH SCRIPT PLEASE
wood

I AM THINKING OF AN OBJECT MADE OF WOOD FOUND IN A HOUSE.
WHAT IS IT. QQ
the floor

NO. TRY AGAIN.
a table

YOU ARE WRONG.  IT IS NOT A TABLE.
it is not a chair

YOU ARE WRONG.  IT IS A CHAIR.
it is a chair

YOU ARE RIGHT.  IT IS A CHAIR.
```

Here is how the computer uses WOOD SCRIPT to carry on its part of this conversation.

The program is initiated by typing "r eliza" and waiting for the response "WHICH SCRIPT PLEASE". The computer will usually type in CAPITAL letters, but it does not matter whether the student uses CAPITAL or lower case letters. After typing the name of the script, two carriage returns must be made to signal the computer that the student has finished. This is true for every input the student makes.

The script is always started at the label START, where some computer statement is usually printed. The computer will then wait for a student response, which when received is stored in the list named INPUT.

C. Keywords, Decomposition, and Reassembly

As the first step in analyzing the list INPUT, the computer scans the sentence for the presence of certain words specified in the script--these are the keywords. CHAIR and TABLE are the keywords in WOOD SCRIPT. If no keywords are found in the input, the system goes to the label NOKEY. This is the case with the answer "the floor". If one or more keywords are discovered in the scanning, the computer takes the keyword with the highest rank according to a priority established in the script (see page 11).

Associated with this selected keyword are a number of decomposition rules, which the computer now applies to the input sentence. The decomposition rules essentially determine whether or not the sentence fits a specific form or pattern. A non-zero positive integer included in these rules stands for that number of words. 0 (zero) stands for "any number of words, including no words at all". As an example, consider the decomposition rule:

```
(0 NOT 1 CHAIR 0)
```

The rule is "any number of words followed by the word NOT followed by one word followed by the word CHAIR followed by any number of words". Thus, the answer "it is a chair" would not fit the rule, while the answer "it is not a chair" would fit the rule. There may be a number of decomposition rules associated with each keyword. The computer takes the first such rule associated with the selected keyword and determines whether or not it fits the sentence. If it does not, it tries the next one. If no rule fits the sentence, the system goes to the label specified at the end of the keyword structure. For the keyword CHAIR, this no decomposition rule label is NOKEY. When it is determined that a rule fits the sentence, the sentence is decomposed into the parts specified by the rule, each part being numbered according to its location in the sentence for future reference. In the example, the sentence "it is not a chair" is decomposed in the following manner:

```
      1       2    3     4      5
( (IT IS) (NOT) (A) (CHAIR) () )
```

The decomposed structure is put in a list named DECOMP.

Associated with each decomposition rule is a reassembly rule that specifies how the decomposed input sentence is to be put back together. It is composed of a series of words and/or numbers. The numbers must be non-zero positive integers that refer to the numbered, decomposed parts of the input. In the example, the reassembly rule:

(IT IS A 4 .)

would reassemble a list to read (IT IS A CHAIR.), since CHAIR is the fourth part of the decomposed input. The name of the reassembled list is SEMBLY.

After the reassembly is completed, the system goes to the label that follows the reassembly rule. In the program part of the script where the labels are located, certain programming functions and commands can be used in many powerful and complex ways. These functions are part of the OPL computer language (Online Programming Language) developed by Professor Weizenbaum, and will be described at length later (see the Appendix). One of their main and most used purposes is to print out a response to the student, which can be either the reassembly list or any other sentence desired. For example, at the label XX in the program section of WOOD SCRIPT, the command

TYPE('YOU ARE WRONG. ', SEMBLY),

causes the computer to type "YOU ARE WRONG." followed by the contents of the list SEMBLY.

After the computer exectues the commands specified in the program section, control is returned to the student and the computer waits for another input sentence in response to what has just been printed. Then the process of searching for keywords and decomposition rules is repeated all over again.

D. The Program

The program of a script lies between (PROG (NAME and END). In this section, OPL functions or commands are used to instruct the computer to perform many different operations, from printing to arithmetic calculations. Over a hundred different functions are listed and described in the Appendix, but the scriptwriter needs to be familiar with only a few of these, especially at first. The most useful functions are listed below with page references to the descriptions.

The section "General Characteristics of Functions" on page A1 is also very important.

Certain rules of format must be followed when writing the program section of a script. The first two lines must be:

```
(PROG     (NAME
          GOTO(POPTOP(DAHIN)).
```

This is referred to as the PROG section. "NAME" is the name of the script, and must consist of no more than six letters and/or numbers, the first character of which must be a letter. In some cases, it is desirable to use more functions between these two lines, but these two must be included for the proper working of a script. The last line of the script must be:

```
          END)
```

This means "close all remaining parentheses", and closes the program section.

The functions of the program are performed sequentially by the computer, a process called executing the program. Several functions are usually needed to cause a certain desired task to be completed. Each function must be separated from the next function by a comma (,), a period (.), or a colon (:). The colon is used only for the functions IF and FOR. The comma means "go on to the next function". The period means "stop execution of the program". This is usually interpreted by the computer to mean "stop and wait for an input from the console" (see

pages 18 and 28 for the exceptions). Each function is usually written on a different line preceded by a tab, but this is not necessary. Different sections of the program may be indicated by labels. These are analogous to bookmarks, and allow the system to be directed to the marked location. Labels must consist of no more than six letters and/or numbers, the first character of which must be a letter. Labels must always be immediately preceded by an asterisk (*), and are followed by a function. When the system goes to a label, it means that the function following the label is executed.

The following section of a program demonstrates some of these points:

```
*XY32     TYPE('WHAT ELSE '),
          NEWTOP('(FREE C), STORE).
*CONSTR   POPTOP(STORE),
          GOTO XY32 .
```

Assume that the analysis of the input caused the system to go to the label CONSTR. The function POPTOP(STORE)--never mind for now what this function does--is executed, and since it is followed by a comma, the function GOTO XY32 is executed. This causes the system to go to the label XY32. It does not matter that GOTO XY32 is followed by a period, since the system transfers to XY32 before encountering the period. Now the function TYPE('WHAT ELSE ') is executed, and since it is followed by a comma, the function NEWTOP('(FREE C), STORE) is executed. The system now encounters the period following this function so exectuion of the program is stopped. Since the computer will usually stop and wait for a student input when a period is encountered, each small section like this should usually contain a printing function somewhere between the input analysis label and the period so that the student will know that the machine is now waiting for him.

The last command preceding a label does not need to be followed by a period. If followed by a comma, for example, the computer will execute that command and then the command following the label, ignoring the label. This allows the computer to enter in the middle of a series of commands, and is sometimes very useful.

Each script must contain the special labels START and NOKEY, since the system assumes that these labels exist. When a script is started from WHICH SCRIPT PLEASE, it begins with the function following the label START. If no keywords are found in an input (and there is no keyword named NOKEYS, see page 12), the system goes to the label NOKEY.

Tabs, spaces, and carriage returns do not affect the program, except to act as separators between characters.

For example, a tab is usually used to separate a label from the function that follows it.

Comments that are not a part of the program may be included by putting them between sets of double slashes (//). They are not considered part of the script when it is being played, but will appear in a printout of the script. For example:

```
//THIS IS THE NOKEY SECTION OF THE PROGRAM//
*NOKEY    TYPE('PLEASE REPHRASE YOUR ANSWER. ').
```

This comment is intended for the human programmer reading a printout of the script. When the computer encounters this section in the course of executing the program, the characters included between the double slashes will be ignored.

E. The Use of Variables

Variables are used in a program to provide an identifier for information that is needed by the script. For example, one variable might be used to keep track of a student's score for the purpose of directing the conversation to harder (or easier) material. Another variable might be used to identify a fragment of text to be printed.

Variable names must consist of no more than six characters, which may be any combination of letters and numbers that starts with a letter. The following are valid variable names: X, COUNT, TXT1, ABCXYZ, SCORE, H32JQ9.

The value of a variable can be a number, a word, or a list of words and/or numbers. A word or number is referred to as a datum, and must consist of six characters or less. A list provides a means for storing information that is longer than six letters, and is usually indicated by parentheses enclosing information. For example:

```
(THERE ARE 3 APPLES)
```

is a list, while any one of the words in the list is a datum. A variable is usually given a value by using an equal sign (=). Use of the equal sign in the program means "replace the value of the left side by the value of the right side". For example, X=X+1 is a valid statement that increases X by 1, even though it is, strictly speaking, algebraically meaningless. This is called an assignment statement. When referring to words or lists in the program as literal elements rather than variables, an apostrophe (') is used immediately preceding the element. For example, A='B sets the value of A to be the letter B, while A=B sets the value of A to be the value of B.

```
TXT1='(THERE ARE 3 APPLES)
```

is an example of a list referred to as a literal element.

F. Input Conventions

When typing input on the console, the signal that the input is completed is two successive carriage returns. A single carriage return does not end the input, so input can continue for several lines if necessary.

If a mistake is made, it may be erased by using the following symbols: # and " erase the single character preceding their use. Multiple use of # or " erase that many preceding characters. For example, the input

```
nw#ow is the tme""ime for all gd me####ood men to come
```

will be read by the computer as

```
now is the time for all good men to come
```

The symbols @ and ? erase the entire preceding part of a current line. The user should continue typing without carriage return. For example, the input

```
today istues@trommor?tomorrow is friday
```

will be read by the computer as

```
tomorrow is friday
```

If the input has gone on several lines and the user wants to erase all of it, he should type $ as the last word of the wrong input and make two carriage returns. The computer will then forget the wrong input and type READY. The correct input can then be typed. For example:

```
this is a dmemonstration of the way to
erase an input when a mistake $

READY
this is a demonstration of the way to
erase an input when a mistake has been
made in a previous line
```

Do not use any of these symbols (# " @ ?) for any purpose other than erasing.

When a single dollar sign ($) is typed as the first word of an input, the system will execute the rest of the input as an OPL program. This may be used by the scriptwriter to check the contents of lists or the value of variables, for example. When execution is completed, the computer will type READY and the user should type the appropriate input.

Chapter 2 -- USING KEYWORDS

A. The Format

The format for writing keywords with associated decomposition rules and reassembly rules in a script is shown below:

```
    KW = keyword
    DR = decomposition rule
    RR = reassembly rule
    :  = more of the same

(KW1      (precedence no.    keycode   (
          (DR1) (RR1) label1
          (DR2) (RR2) label2
            :     :     :
          (DRn) (RRn) labeln
          ) no DR label  ))
(KW2      //same structure//
  :             :
(KWm      //same structure//
(PROG     (NAME
          GOTO(POPTOP(DAHIN))
          //the program section//
          END)
```

B. Keycodes

Immediately preceding the third left parenthesis of each keyword is the keycode. This is a word of six letters or less that must be unique for every keyword in the script since the system uses it to identify individual keywords. The keycode is usually mnemonic, such as the keycode CHR for the keyword CHAIR in WOOD SCRIPT (see page 2), but it can actually be any unique combination of letters. The keycode may be the same as the keyword if the keyword is less than six letters.

C. Precedence Numbers

Keywords may be ordered according to their importance in a script. This is done by including a precedence number following the second left parenthesis in the keyword structure. This number may be any integer less than 262144. If the number is omitted, the system assigns zero as its precedence number. For example, in WOOD SCRIPT, the keyword CHAIR is more important than the keyword TABLE, since CHAIR is the right answer. Therefore, CHAIR is given a higher precedence number (10) than TABLE (0 since omitted).

D. NOKEYS

Sometimes it is helpful to use decomposition rules on an input that does not have any keywords. For example, when anticipating numbers within a certain range in the input, there are too many possibilities to have a keyword for each. See page 14 for a description of the detection of numbers with decomposition rules. Decomposition rules may be used on such an input by including the keyword NOKEYS in the script as a regular keyword. If no keywords are found in the input, the system will look for the keyword NOKEYS. If it is present, it will be treated as a regular keyword. If it is not present, the system will go to the label NOKEY. The following keyword structure is an example of the use of NOKEYS.

```
(NOKEYS    (NOKEYS    (
           (0 DESK 0) (2) XX
           (0 FLOOR 0) (2) XX
           ) NOKEY ))
```

The no decomposition rule label should never be the label AGAIN (see page 53), since this will result in an infinite loop. This label should usually be NOKEY.

E. The Keystack

When the input is processed, the system scans it from left to right looking for keywords. The keywords that are found are stored in the order of their occurrence in the input in a list called the keystack. If a keyword is used more than once in an input, it will be put on the keystack as many times as it is used. See the section "Keyword and Sentence Analysis Functions" on page A13 for a description of the functions that can be used to manipulate the keystack.

F. Keyword Phrases

Keywords may be single words, or they may be phrases. Integers, written as (*N), may be used with the same meaning as in decomposition rules, that is, they stand for that number of words. "Any number of words" is written as (*0). The only requirement for a keyword phrase is that it start with a word and not a number. For example, the keyword phrase:

```
(SALT (*1) PEPPER (*0) SUGAR          ( etc.
```

is read as "the word SALT followed by one word followed by the word PEPPER followed by any number of words followed by the word SUGAR". The keyword phrase must start with a word.

G. Substitutions

Keywords that have the effect of making substitutions in the input may be included in a script. For example, if a number of different words have the same meaning, use of substitutions will save duplication of effort in writing keywords and decomposition rules. There are several forms of the substitutions. The words TURF, GRASS, and FIELD are used as examples.

```
(TURF = GRASS)      simple substitution
```

When TURF is found in the input, it is replaced by GRASS. GRASS is not treated as a keyword and is not put on the keystack. When decomposition rules are applied, the substitution has already been made, so GRASS must be used in place of TURF where needed. This form may be used to delete words or symbols from the input by leaving GRASS blank. For example, (, =) would remove all commas from the input.

```
(TURF = GRASS .)      substitute and rescan
```

TURF is replaced by GRASS, and the system checks to see if GRASS is a keyword. If it is, it is put on the keystack. This may be used to allow the computer to accept common misspellings, such as (FEILD=FIELD.). If both (TURF=GRASS.) and (GRASS=FIELD.) are included in the keyword section, FIELD will be put on the keystack if it is a keyword.

```
(TURF = GRASS(40 (FIELD)))      substitute and treat as the
                                keyword FIELD
```

TURF is replaced by GRASS and FIELD is put on the keystack with the precedence number 40. If FIELD is the highest ranking keyword, the actual keyword structure for FIELD is found and the decomposition rules applied. If a precedence number is not included, it is made 0 (zero).

Chapter 3 -- USING DECOMPOSITION RULES

A. Different Forms

The following forms may be used as single elements in decomposition rules for the described purpose.

(*GO GOES GOING)

Any of the three words GO, GOES, GOING used in this place in the input will fit this part of the decomposition rule. Any number of words may be used following the asterisk (*).

(/MALE FAMILY)

This form is called a tag list. A word will fit this part of the decomposition rule if it is a part of the categories MALE and FAMILY. The categories are specified in the keyword section by a description list of the keyword. For example the keyword (BROTHER DLIST(/MALE FAMILY SIBLNG)) identifies BROTHER as belonging to the categories MALE, FAMILY, and SIBLNG. These words are the tags, and must consist of six letters or less. They do not need to be mnemonic, although these are. In order to fit part of a decomposition rule of this form, the tags on the DLIST of the keyword must include all the tags on the tag list in the decomposition rule.

(COND expression)

This form can be used to detect numbers. "Expression" may be a Boolean expression. The Boolean operators E, LE, GE, L, and G may be used. For example, (COND L 7 AND GE 5) means "a number less than 7 and greater than or equal to 5".

Lists may be detected if "expression" is the word LIST. A list is indicated in an input by parentheses, and is treated as a single element.

Numbers that are within a certain range of a specified number may also be detected. In this case, "expression" should be EPS F1 F2, where F1 and F2 are numbers. F2 specifies the percentage range around F1. A number will fit this part of the decomposition rule if it is within (either plus or minus) F2 times F1 of the number F1. For example, (COND EPS 4 .1) means any number within .1*4=.4 of 4, or any number between 3.6 and 4.4 .

$

The dollar sign is used to indicate "any number of letters, but at least one letter" and may be used as a prefix, as a suffix, or in the middle of a word. For example:

MEAN$	any word which begins with the letters MEAN (e.g., meant, meanwhile)
$BALL	any word which ends with the letters BALL (e.g., baseball, football)
$RING$	any word which contains the letters RING (e.g., earrings, fringe)
DO$ING	any word beginning with the letters DO and ending with the letters ING (e.g., donothing, doubting)

The following decomposition rule demonstrates the use of these forms:

```
(0 (COND E 3) (/MALE FAMILY) 0 (*GO GOES GOING) TO$ 0)
```

It means "any number of words followed by a number equal to 3 followed by a word which belongs to the categories MALE and FAMILY followed by one of the three words GO, GOES, or GOING, followed by a word beginning with the letters TO followed by any number of words".

B. Ordering

When the highest ranking keyword in an input is found, its decomposition rules are applied beginning with the first continuing to the last or until one is found to fit. Thus, an ordering of decomposition rules will help in analyzing an input. Generally, the most important rules should be placed first, since these will be the first applied. Also, the most general rules (the ones that will fit the most inputs) should be placed last. In particular, if the rule (0) is used, it should always be placed last, since it will fit any input.

Chapter 4 -- USING REASSEMBLY RULES

A. General Uses

There are two general uses of reassembly rules. The first use is to form a sentence from the input to be printed back by the computer. This use is shown in the following decomposition rule and reassembly rule pair:

```
(0 I 0 LOVE 0) (WHY DO YOU 4 5)
```

To cause the reassembly list to be printed requires a print command in the program section.

The second general use of reassembly rules is to extract information from the input, but not necessarily to print back to the student. The following decomposition rule and reassembly rule pair shows how the value of X may be extracted from the input:

```
(0 X 0 EQUALS 1 0) (5)
```

The reassembly list will contain the one word that immediately follows the word EQUALS in the input, which is presumably the value of X.

B. Decomposed Words ($)

When the dollar sign is used as part of a decomposition rule (see page 15), the reassembly rule may be used to extract all or parts of the word. The word that is decomposed is split into parts by the dollar sign. For example, the decomposition rule (DO$ING) would separate the word DOUBTING into three parts: DO UBT ING . These parts may be referred to in the reassembly rule by a list of the form (N1,N2). N1 is the integer that stands for the place of the word in the decomposed input. N2 is the integer that stands for the part of the decomposed word. For example, if the decomposition rule

(0 DO$ING 0 $RING$ 0)

were applied to the following input

HE IS DOUBTING THE EFFECTIVENESS OF THEIR STRINGENT MEASURES

the different parts of the decomposed words could be referred to by the following pairs of numbers:

(2,1)	DO
(2,2)	UBT
(2,3)	ING
(4,1)	ST
(4,2)	RING
(4,3)	ENT

If it is desired to put parts of the separated word back together, a dollar sign should be used in the reassembly rule between the pairs of numbers that refer to the parts. For example, to put back together the first two parts of the decomposed word ST RING ENT , the reassembly rule should be:

((4,1)$(4,2))

The reassembly list would be (STRING). If it is desired to refer to the whole word that has been decomposed, a single integer referring to the place of the word in the decomposed input (i.e. N1) should be used alone. Thus, the reassembly rule (4) would result in a reassembly list (STRINGENT).

Chapter 5 -- USING THE PROGRAM

A. Execution of the Program

The result of the analysis of an input is usually the specification of a label referring to a place in the program section of the script. This label is put on the top of a list called DAHIN (German for "there") by the system. The system transfers control to the program section by executing the program. This means that the functions in the program are executed sequentially beginning with the first one. Since the first function in the program section is usually

```
GOTO(POPTOP(DAHIN)).
```

execution of the program will cause the label on top of DAHIN to be taken off and the system to go to that label.

When a period is encountered in the program section, execution of the program is stopped. This is usually interpreted by the system to mean that it should wait for another input. However, there are certain exceptions. When execution is stopped, the system checks the list DAHIN. If it is empty, and the variable KEY is zero (see page 28), the system waits for an input. If there is a label on top of DAHIN, the program is executed again, which has the effect of sending the system to that label. For example:

```
NEWTOP('START,DAHIN).
```

would cause the system to go to the label START, the same effect as the function: GOTO START. If the system finds a list on top of DAHIN, it assumes that this indicates a "provisional transfer", which is useful when more than one script is used (see "Changing Scripts" on the next page.) The system assumes that the list contains a label and the name of a script. This could be done, for example, by:

```
NEWTOP('(START WOOD),DAHIN) .
```

In this case the system checks to see if the bottom element, e.g. WOOD, is the name of the script that is being played. If it is, the list on DAHIN is replaced by the top element, e.g. START, and the program is executed again. If it is not the same, the system treats the situation as if DAHIN were empty. Thus, when a list is put on DAHIN, it has the effect that the system will go to the label on the top of the list only when the script whose name is on the bottom of the list is being played.

This checking procedure is shown diagrammatically along with other aspects of the operation of the system in the "Flow Chart of the System" on page A30.

B. Group Areas

The ELIZA system provides a number of group areas where scripts may be located in a playable condition, even though only one script may be played at a time. The name of each group area is SA(N) where the groups are distinguished by the integer N. The control script (see Chapter 6, which begins on page 35) is always located in group area SA(0). The script specified in response to WHICH SCRIPT PLEASE is read or brought into group area SA(1). Other scripts may be brought into other areas, either by using the function SCRIPT or by using the control script. The value of the variable SCRPN(N) is the name of the script in group area N. If there is no script in a particular area, the value is 0 (zero). The name of the keystack of each group area is KA(N). The variable GROUP is always equal to the number of the presently active group area where the script that is being played is located. Thus if the scriptwriter wants to refer to the presently active keystack, he should write KA(GROUP). The number of group areas is indefinite, but if more than three or four are used (depending on the size of the scripts), the available space in the computer core memory will be exhausted.

C. Changing Scripts

To change a script means to make a different script active by switching control of the conversation to it. Scripts may be changed for a number of reasons. The scriptwriter may wish to change the subject of discussion to something covered in another script. With the new script will come a new set of keywords and decomposition rules. It is also easier to program and correct a number of small scripts rather than one large one.

In order to change scripts, four things must be done if the scriptwriter is going to program the change himself. (An alternative method using the control script is described on page 48.) (1) The script must be read into the appropriate group area by using the function SCRIPT (see page A24). DO NOT read a script into the group area where the presently active script is located! (2) Set GROUP equal to the number of the group area where the new script is located. (3) Put the label where the system should go in the new script on top of DAHIN. (4) Include a period to tell the system to stop executing the program. For example:

```
SCRIPT(2,'ELEVTR),
GROUP=2,
NEWTOP('START,DAHIN) .
```

This series of commands causes the computer to read the script named ELEVTR into group area SA(2), to transfer control to area 2 (when the period is encountered), and to

go to the label START in the program section of ELEVTR SCRIPT. The order of the three commands is not important.

D. Variables and Scripts

When the scriptwriter is using a number of scripts, it is important to distinguish the variables "known" only to one script from those "known" to all scripts. For example, the variable X might be used to keep track of the number of wrong answers the student gives in each script. If X is used for this purpose in different scripts, the system should know that there are in effect different Xs. The variable Y might be used to keep track of the total number of wrong answers in all scripts, and the system should know that when Y is referred to in different scripts, it is the same Y. The functions that enable the scriptwriter to distinguish these cases are COMMON and OWN or OWNLIST. COMMON should contain the names of all the variables that should be known to all scripts. For example:

```
COMMON(Y,STOUT,STOUTN,PLACE,SCRNAM),
```

tells the computer that the variables Y, STOUT, STOUTN, PLACE, and SCRNAM are to be known to all scripts in common. The common variables are usually, but not always, listed in the first script to be played.

OWN and OWNLIST provide different ways to specify variables to be "bound" or known only to the script that includes the statement. OWN is used when specifying the variables without giving them values. For example:

```
OWN(X,OLDLAB,OLDNAM.)
```

tells the computer that the variables X, OLDLAB, and OLDNAM are to be known only to the script in which the OWN statement occurs. The last variable must be followed by a period (e.g. "OLDNAM."). OWNLIST is used to give values to some of the variables, as well as making them local to the script. For example:

```
OWNLIST(X,OLDLAB,OLDNAM.) AND (J=1) (MTA='(A LIST)),
```

tells the computer that the variables X, OLDLAB, OLDNAM, J, and MTA are to be known only to the script in which the OWNLIST statement occurs. It also tells the computer to set the value of J equal to 1 and to set the value of MTA equal to the list (A LIST). All the variables not assigned values by OWNLIST should be included in the first set of parentheses following OWNLIST. Again, the last variable in the list must be followed by a period. Each list following the AND should include an assignment statement for a variable.

COMMON, OWN, and OWNLIST differ from other OPL functions in that they are executed before the program section is executed. This means that it does not matter where they are located in the program section, for they are executed when the script is read in and before control is transferred to it. Their values may then be changed in the script as necessary.

E. Subscripted Variables

Subscripted variables provide a simple notation for vectors and matrices. If the variable is to be a one-dimensional array or vector, the form is:

```
B='(ARRAY)
```

For multi-dimensional arrays, the maximum values must be given in the form:

```
C='(ARRAY 10 9 15)
```

This creates C as a 10 by 9 by 15 array. There is no limit to the number of dimensions or the maximum values, but neither can be infinite. The subscripted variable is now referred to by statements of the following type:

```
B(7) = C(4,5,1) + 10
```

This command says "set the value of the seventh component of the vector B equal to the value of the element located at position 4,5,1 of the array C plus the number 10. The subscripted variable may appear in any place that a non-subscripted variable may appear, including as an argument of a function. The subscripts may be any OPL program that results in a numerical result. For example:

```
X = C(SQRT(X), X/Y+1, 3)
```

is a valid statement.

F. Context Awareness--The TABLE Mechanism

Description

It is often useful when writing scripts to be aware of which question or statement the person who is playing the script is responding to. It is also useful to use the same keyword and decomposition rules in analyzing responses to more than one question, where the computer response is determined by which question was asked as well as by which decomposition rule fits. A simple example (illustrated below) is a script that asks a series of questions expecting yes or no answers, and selects the next question on the basis of the preceding answer.

The ELIZA mechanism that makes this "context awareness" possible involves additions at two points in the script. The first is in the program section, at the point where the question is asked. The other is in the decomposition rules that will fit possible answers to the question.

In the program section, use is made of a list whose name is TABLE, which is known by the ELIZA system. A list containing pairs of words is put on the top of TABLE. The first word of each pair is called a tag, which can be any combination of six letters or less. The tag will be used for detection in the decomposition rules. The second word of the pair must be a label in the program section of the script. This label is the location in the program to which the system will go if the tag is detected. For example, in TEST1 SCRIPT (page 25) the tags and labels are put on the list TABLE following the label START:

```
NEWTOP('(YYY B NNN C O'E D), TABLE)
```

YYY is a tag and B is the label associated with it; NNN is a tag and C is the label associated with it; O'E is a special tag which means otherwise (its use is discussed below) and D is the label associated with it.

In the decomposition rules that match possible statements of the person who will play the script, tags are put in description lists associated with the decomposition rule. This can be done by including the word DLIST followed by the contents of the description list in parentheses anywhere within the rule. For example, in TEST1 SCRIPT, under the keyword YES, the decomposition rule is written:

```
(0 YES 0 DLIST(YYY))
```

This associates a description list that has YYY in it with the decomposition rule (0 YES 0).

The basic operation of the TABLE mechanism is to compare the list on the top of the list TABLE (which is given the name ELBAT) with the description list (if any) associated with a decomposition rule to see if there are any identical tags, and if not, to see if there is any 0'E tag on that list. This is done following an input analysis. An input analysis will always yield one of the following three results:

1. A keyword is found, and a decomposition rule fits the input. When a decomposition rule fits an input, the reassembly rule is performed. Then ELIZA checks to see if the decomposition rule has a description list, and if it does, ELIZA checks to see if any of the tags on the description list are also on ELBAT. If it finds a tag that is on both, the system goes to the label that is associated with the tag in the word pair on ELBAT. If there are no matching tags, ELIZA checks to see if there is an 0'E tag on ELBAT. If there is an 0'E tag, the system goes to the label associated with the 0'E tag in the word pair on ELBAT. If there is no 0'E on ELBAT, the system goes to the label following the reassembly rule, as if the TABLE mechanism had not been used at all. There is an important exception to this checking for the 0'E tag. If the precedence number of the keyword is greater than or equal to 10000, the situation is treated as if there were no 0'E tag on ELBAT. This permits the scriptwriter to override the TABLE mechanism when desired.

2. A keyword is found, but no decomposition rule fits the input. ELIZA checks to see if there is an 0'E tag on ELBAT, and if there is, the system goes to its associated label. If there is no 0'E tag on ELBAT, the system goes as usual to the label specified at the very end of the keyword structure--the "no decomposition rule" label. Again there is an important exception. If the precedence number of the keyword is greater than or equal to 10000, the situation is treated as if there were no 0'E tag on ELBAT.

3. No keyword is found. ELIZA checks to see if there is an 0'E tag on ELBAT and, if there is, the system goes to its associated label. If not, the system goes to the keyword NOKEYS (see page 12), if it exists; otherwise the system goes as usual to the program label NOKEY.

In all cases, after TABLE is checked, ELBAT is taken off, so that the tags and labels put on TABLE are there for only one input. This usually empties TABLE.

In the case of labels such as NOKEY, however, it is usually desired to keep the list on TABLE. Since ELBAT is the name of the last list taken off TABLE, it can be put back on again by:

```
NEWTOP(ELBAT,TABLE)
```

If the scriptwriter wants to process an input without using TABLE, he should write:

```
LIST(ELBAT),
```

so that the system will not get confused with a previous list.

The tags used are completely arbitrary. The only requirement is that the tag used in a list on TABLE must be the same as the tag in the description list of the decomposition rules the scriptwriter wants to detect.

If a list of labels is used in place of a label, the system will go to the first label on the list the first time its tag matches, the second label the second time its tag matches, and so on. When the last label on the list is reached, it will be used for all succeeding times that the tag matches. For example, the following command could be written:

```
NEWTOP('(YYY B NNN C O'E (D E F)),TABLE)
```

When the O'E tag is invoked, the system will go to label D the first time, label E the second time, label F the third time, label F the fourth time, and so on. This technique can be used to avoid trapping the student when the script repeatedly fails to recognize his answer. When a label is taken off such a list, the system also takes the first occurrence of that label off any other lists on ELBAT. For example, if the list

```
(YYY B NNN (C D E F) O'E (D E F))
```

is put on TABLE and the O'E tag is invoked following an input, the system will go to the label D the first time and the list will be:

```
(YYY B NNN (C E F) O'E (E F))
```

Example

```
TEST1 SCRIPT

(YES      (YES      (
          (0 YES 0 DLIST(YYY)) (THAT'S GOOD.) A
          ) NOKEY))
(NO       (NO       (
          (0 NO 0 DLIST(NNN)) (THAT'S TOO BAD.) A
          ) NOKEY))
(PROG     (TEST1
          GOTO(POPTOP(DAHIN)).
*START    NEWTOP('(YYY B NNN C O'E D), TABLE),
          TYPE('HAVE YOU STUDIED QUANTUM MECHANICS.QQ ').
*A        TYPE(SEMBLY),
          QUIT(0).
*B        TYPE('DID YOU ENJOY QUANTUM MECHANICS.QQ '),
          LIST(ELBAT).
*C        TYPE('WILL YOU STUDY QUANTUM MECHANICS.QQ '),
          LIST(ELBAT).
*D        TYPE('PLEASE ANSWER EITHER YES OR NO. '),
          NEWTOP(ELBAT,TABLE).
*NOKEY    TYPE('PLEASE REPHRASE YOUR ANSWER. '),
          NEWTOP(ELBAT,TABLE).
          END)
```

conversation one:

```
r eliza
W 1518.7
WHICH SCRIPT PLEASE
test1

HAVE YOU STUDIED QUANTUM MECHANICS. QQ
maybe

PLEASE ANSWER EITHER YES OR NO.
yes

DID YOU ENJOY QUANTUM MECHANICS. QQ
yes

THAT'S GOOD.
R 3.127+1.983
```

conversation two:

```
r eliza
W 1519.6
WHICH SCRIPT PLEASE
test1

HAVE YOU STUDIED QUANTUM MECHANICS. QQ
no

WILL YOU STUDY QUANTUM MECHANICS. QQ
no

THAT'S TOO BAD.
R 2.187+1.624
```

TEST1 SCRIPT and the conversations that follow it demonstrate simply some of the uses of the context awareness ability of the TABLE mechanism. Conversation one will be discussed, and conversation two will serve as an additional example. In conversation one, the script is called, and the system begins at the label START. A list containing the tags and labels in pairs is put on the top of TABLE, as described above. The question "HAVE YOU STUDIED QUANTUM MECHANICS. QQ" is asked, and ELIZA waits for a response. The answer "maybe" does not have a keyword of the script in it. ELIZA checks and finds an 0'E tag on ELBAT, the top list on TABLE. The system goes to the label D, associated with the 0'E tag on ELBAT. In the process, ELBAT is taken off TABLE, so TABLE is now empty. Following the instructions at label D, the computer prints "PLEASE ANSWER EITHER YES OR NO.", puts the list of tags and labels (which is ELBAT) back on TABLE, and waits for a response. This time the response is "yes", which has the keyword YES. The decomposition rule (0 YES 0 DLIST(YYY)) is found to fit. Since this has a description list, ELIZA checks and finds that the tag YYY is on both the description list and ELBAT, the top list on TABLE. The system goes to the label B, associated with YYY on ELBAT. In the process, ELBAT is taken off TABLE, so TABLE is now empty. Following the instructions at label B, the computer prints the question "DID YOU ENJOY QUANTUM MECHANICS. QQ" and waits for a response. The response "yes" fits the same decomposition rule, but since there is no list on TABLE this time, ELIZA then performs the reassembly rule and goes to the label A, where SEMBLY is printed, "THAT'S GOOD."

Technique

The tags in the description list of a decomposition rule are checked every time the rule is the one that fits the input. The tags on TABLE, however, are put on only when certain points in the program section of the script are

reached by the system, and they stay on for only one input. The technique for using this mechanism effectively in scripts more complicated than TEST1 SCRIPT is to have only one specific tag in the description list for any one particular decomposition rule. Then, for each decomposition rule that would fit a possible answer to a question, the tag that is in its description list and an appropriate label for the system to go to if the input is fitted by the decomposition rule, are put as a pair on TABLE. This point is the essence of the context awareness, since the scriptwriter knows what questions or statements will be printed out, and what responses he expects. TEST1 SCRIPT shows this technique in elementary form. In answer to the initial question "HAVE YOU STUDIED QUANTUM MECHANICS. QQ", (0 YES 0) and (0 NO 0) are the important decomposition rules. Their associated tags are YYY and NNN respectively. If (0 YES 0) fits the input, the system should go to the label B. Thus, label B is associated with the tag YYY in a word pair put on TABLE. Likewise, the system should go to the label C if (0 NO 0) fits the input, so label C is associated with the tag NNN in a word pair put on TABLE. If any other form of input is typed, the system should go to label D, so label D is associated with the tag 0'E in a word pair put on TABLE. All of this is accomplished by the one NEWTOP statement, which puts all of these word pairs in one list on the top of TABLE. If there are more decomposition rules that the scriptwriter wishes to detect if they fit the input, this process of putting tags and labels on TABLE is simply extended by putting tags in description lists in the decomposition rules and corresponding pairs on TABLE. Another important example is the case where one decomposition rule is ambiguous in different contexts, i.e., where the scriptwriter may expect the same input to have entirely different meanings. By putting different labels with the tag that is in the description list of the decomposition rule on TABLE in the different contexts, these contexts can be distinguished in the decomposition rule.

One further use of the 0'E tag is worth mentioning. The system will go to the label associated with an 0'E tag if there are no matching tags on the description list and TABLE, regardless of whether or not a keyword is found in the input or a decomposition rule is found to fit. Thus, an 0'E tag can be used to go to a specific label, regardless of the content of the input. Examine the following section of the program part of a script:

```
        NEWTOP('(0'E FF), TABLE).
*FF     TXTPRT('(THAT'S INTERESTING),0),
```

The period following the NEWTOP function means to wait for an input from the console. ELIZA will then go through the normal procedure of keyword and decomposition rule analysis (including substitutions, which may be important), but since

there is only an O'E on TABLE, the system will go to the label FF in the program section regardless of what the input is (unless the precedence number of a keyword in the input is greater than or equal to 10000). ELIZA will print "THAT'S INTERESTING" and continue with the program.

G. Sentence Analysis

Certain of the OPL functions can be used to analyze the contents of a list in the same manner that the system analyzes input. These are usually needed only for special purposes, since the keyword structure usually provides sufficient analysis. Readers unacquainted with these functions should omit the following summary and skip to the section on "Counters" below. The functions are described in detail in the "Keyword and Sentence Analysis Functions" on page A13.

Within the program of a script, the functions KEY, WASKEY, and HIRANK can be used to locate a desired keyword structure. The system may then be instructed to apply this keyword structure, consisting of decomposition rules, reassembly rules, and labels, to a certain list. This may be done in the following way. When the system encounters a period in the program, it stops the execution. In addition to checking the list DAHIN (see page 18), the system also checks the variable KEY. If KEY is equal to zero, the usual case, the system will wait for an input. However, if KEY is not equal to zero, the system assumes that KEY is the list that contains the keyword structure, and that the variable EXP is the list to which the structure should be applied. The system then processes the list EXP in the same way that it processes input.

Input may be reprocessed in this manner. If the keyword structure first applied by the system yields no useful information, the system can be instructed to apply the next highest ranking keyword on the keystack. The following program section demonstrates this procedure.

```
Z=0,
HIRANK(KA(GROUP),0,-1),
Z=HIRANK(KA(GROUP),0,1),
IF Z .E. 0
  THEN NEWTOP('NOKEY,DAHIN).
  ELSE KEY=Z, EXP=INPUT. :
```

When using the TABLE mechanism, the defined function KKK may be used to do this (see page A29), instead of this program section. This is also the purpose of the label AGAIN in the unit discussion scripts (see page 53).

The functions MATCH and ASSMBL may be used to apply decomposition rules to lists and to reassemble lists from a decomposed list. For example:

```
        IT=MATCH('(0 NOT 1 CHAIR 0),INPUT,LIST(PMOCED)),
          //DECOMPOSED LIST IS PMOCED//
        IF IT .E. 0 THEN GOTO NEXTDR :
        ASSMBL('(IT IS A 4 .),PMOCED,LIST(YLBMES)),
          //REASSEMBLY LIST IS YLBMES//
        GOTO XX .
*XX     TYPE('YOU ARE WRONG. ',YLBMES).
```

This program section decomposes and reassembles in the same way that the first decomposition and reassembly of the keyword CHAIR in WOOD SCRIPT is performed (see page 2).

H. Loops

When a scriptwriter wishes to program a certain operation a number of times, it is usually bad to write the program section for each separate use. This can take up a lot of space in the script and is laborious work if the number is large. A programming technique called a loop may be used to avoid these problems. The loop uses the same program section over and over again the required number of times. The last statement in the section sends the system to the top of the loop to perform the operation again. Within the section there should be a test that provides a way out of the loop. That is, the loop will continue until the test is met; then the system will go somewhere else. The following program section demonstrates the use of the loop to print every other item in the list named TAB. The sequence reader functions are described on page A11.

```
        S=SEQRDR(TAB),          //INITIALIZATION//
*PQ     XX=SEQLR(S),            //TOP OF LOOP//
        IF XX .E. 'NIL          //TEST//
          THEN GOTO PR :
        XX=SEQLR(S),
        TYPE(XX),
        GOTO PQ .               //BOTTOM OF LOOP, GO TO TOP//
*PR     TYPE('END OF EXAMPLE ').
```

The loop can be used to perform operations a number of times where the number is not known, as in the above example. The scriptwriter should be sure that the test will be met at some time, or the result will be an "infinite loop". The loop may also be programmed by using the FOR statement (see page A20).

I. Counters

Counters may be used to keep track of the number of times a certain operation is performed, such as right answers, wrong answers, or the number of times through a loop. The counter is a variable that is increased by a certain number (usually one) each time the operation is performed. Counters must be set to initial values before they are used.

The following program section demonstrates the use of the variables R and W to keep track of right and wrong answers. The total score is computed as a percentage score where one-quarter credit is subtracted for each wrong answer.

```
*START    R=0, W=0,                  //INITIALIZATION//
*RTA      R=R+1 .                    //A RIGHT ANSWER//
*WRA      W=W+1 .                    //A WRONG ANSWER//
*SCORE    S=((R-W/4)/(R+W))*100 ,
          TYPE('YOUR SCORE IS ' S).
```

J. Lists

ELIZA is based on a list processing computer language called SLIP ("Symmetric List Information Processor") developed by Professor Weizenbaum. All inputs, keyword structures, and outputs, for example, are treated as lists. A list consists of a series of cells that are linked together. Each cell contains six characters, some of which may be blanks. If a word that is stored in a list is longer than six letters, the word is stored in two or more cells. For example, the list:

```
T='(THE WEATHER IS BEAUTIFUL.)
```

is stored in the computer as a series of cells, where - indicates a blank:

```
THE---
WEATHE
R-----
IS----
BEAUTI
FUL---
.-----
```

Each cell is marked as to whether or not it is a word or part of a word (see page A15), so that when the printing functions are used, the list will be printed in its original form. The extra blanks, if any, are not printed. Each cell on the list is stored in a certain location in the computer memory called the machine address of the cell. This may be obtained for a cell by using the functions MADOBJ or SEQPTR (see pages A6 and A12, respectively).

When a list is placed on another list, the first list becomes a sublist of the other list. In the memory of the computer, only the name and location of the sublist are put in a single cell in the other list. The sublist remains in its previous location. The name and location of the sublist serve as a pointer to it. For example, when

```
NEWTOP('(I THINK THAT),T)
```

is written, the lists are stored in the computer as:

```
sublist=========|-----
THE---          THINK-
WEATHE          THAT--
R-----
IS----
BEAUTI
FUL---
.-----
```

Sublists themselves may contain sublists, and this may continue to any desired depth.

The preceding considerations about cells are important when using functions that operate on individual cells in a list, such as the sequence reader functions (see page A11).

K. Special List Problems

There are a couple of special aspects of lists that have caused problems to unwary scriptwriters. Consider the following program section:

```
*M13Q     X='(THIS LIST),
          TYPE(X),
          Y=X,
          MTLIST(Y),
          GOTO M13Q .
```

The list X is created by an assignment statement and stored in the computer. The TYPE function causes THIS LIST to be printed. The command Y=X gives Y the value of this same list, but does not copy it. That is, X and Y point at the same list. Thus, after MTLIST(Y) is performed, both X and Y point to the same empty list, even though there was no MTLIST(X). Also, when the system goes back to label M13Q, it looks as if X will be reset to (THIS LIST). However, this list is the one that has been emptied, so that X still points to the empty list. Thus, when the TYPE function is executed this time, an empty line is printed, which might not have been the desire of the scriptwriter.

Another problem is concerned with list erasures. In order to provide more available space for the program, the system periodically does some automatic clearing up of space that is not needed. Lists are erased when they are not the value of a variable, not a sublist of another list, and not pointed to by a function. Consider the following program section:

```
*JKL      X='(FIRST LIST),
          X='(SECOND LIST),
          GOTO JKL .
```

When the value of X is changed from the FIRST LIST to the SECOND LIST, the FIRST LIST is no longer pointed to by anything. By the time that the system returns to the label JKL, the FIRST LIST may have been erased, and the storage area to which X would point filled with something else. This problem may be avoided by using different variables to refer to different lists if the lists are to be used more than once.

These problems do not apply to variables whose values are data, rather than lists.

L. Description Lists

A description list is a list that is "associated" with another list. This means that the description list can be referred to by referring to the other list. When the other list is printed, however, the description list is not printed. Thus, description lists can be used to store information in a sort of "invisible" way since it will not be part of any output.

There are two ways to make a description list. The first is to use the function MAKEDL (see page A9). The second is to use the word DLIST followed by the description list in the other list. For example:

```
G='(THE OTHER LIST),
MAKEDL('(THE DESCRIPTION LIST),G),
```

has the same effect as:

```
G='(THE OTHER LIST DLIST(THE DESCRIPTION LIST)),
```

The DLIST may be used at any position in the other list. The functions that are concerned with description list are described on page A9.

M. Paired Information

In most of the description list functions, it is assumed that the description list is made up of pairs of data. In the function VAL (see page A9), this assumption is made, even though it does not refer to a description list. The ability to refer to information stored in pairs is sometimes helpful in scriptwriting. For example, consider the situation where the scriptwriter wants the system to go to one of a certain number of labels, depending upon the value of a variable. Assume that the value is to be a number (spelled out) between one and five. In the following program section, the variable AA has been set to one of the values one, two, three, four, or five elsewhere in the course of the conversation. The list M4 contains values and labels as pairs of data.

```
M4='(ONE XX TWO YY THREE XX FOUR M13Q FIVE START),
GOTO(VAL(AA,M4)).
```

This technique is much better and faster than using a series of IF statements to determine the value of the variable AA.

N. Long Printouts

When there is a large amount of material to be printed, the function PRTLC ("print lower case") should be used instead of TYPE or TXTPRT. PRTLC causes a file that is stored on the disk to be printed. This means that the material to be printed can be stored on the disk rather than in a script in memory where there is less available space. PRTLC also has the advantage that more characters can be used: all the characters on the teletypewriter including upper and lower case letters. The function PRTUC ("print upper case") can be used to print a file of only upper case characters. The formats of PRTLC and PRTUC are described on page A23.

Chapter 6 -- USING THE CONTROL SCRIPT

A. The Control Script

The control script is a very powerful device for increasing the computer's versatility as a tutor. When the control script is used, it is called into group area SA(0) and ordinarily remains there throughout a discussion. The control script does the following:

1. Automatically adds often-used keywords (e.g. many variants of "yes" and "no") to every script played during the discussion (Section B).
2. Allows the student to control the conversation by means of various interruptions (Sections C, D).
3. Provides machinery for manipulating many small, easily-written "unit" scripts that carry on the discussion (Sections E, F, G, H).

These services of the control script are described in detail in this chapter.

B. Control Script Keywords

The keywords of the control script are automatically added to each script while it is being played. The scriptwriter must use the proper method for bringing in a new script. This method is described in section F (1) on page 48. Certain equivalences or substitutions for the words yes, no, and not are included in the control script to save programming for the scriptwriter. The following list shows the equivalences contained in the control script.

```
(DONT = DO NOT .)
(WON'T = WILL NOT . )
(ISN'T = IS NOT . )
(AREN'T = ARE NOT . )
(RIGHT = RIGHT ((YES)))
(TRUE = YES.)
(FALSE = NO.)
(INCORRECT = NO.)
(CORRECT = YES.)
(WRONG = NO.)
(O K = YES .)
(SURE = YES.)
(OK = YES.)
(O.K. = YES.)
(OKAY = YES.)
(OF COURSE NOT = NO.)
(CERTAINLY NOT = NO.)
(OF COURSE = YES.)
(CERTAINLY = YES.)
```

```
(YEAH = YES.)
(NOPE = NO.)
(DON'T = DO NOT.)
(DOESN'T = DO NOT.)
(NOT DLIST(/NOT.))
(CAN'T = CAN NOT.)
(CANNOT = CAN NOT.)
(DIDN'T = DO NOT.)
```

The substitutions ending in periods mean, for example, "substitute yes for yeah in the input and resume checking to see if yeah is a keyword". The DLIST(/NOT) indicates that the word belongs to the category indicated by NOT, and any word in the category will fit a part of a decomposition rule that is (/NOT). (See the sections on "Substitutions", page 13, and "Different Forms", page 14.)

The following keyword sections for yes and no are contained in the control script, and thus included in all scripts while being used.

```
(YES      (YES     (
          (0(/NOT) 0 (*RIGHT YES) 0 DLIST (CONTRL NO NOQ))
          (0 (*YES RIGHT) 0 DLIST(CONTRL YES YESQ))
          ) AGAIN ))
(NO  DLIST(/NOT) (NO      (
          (0 (/NOT) 0 NO 0 DLIST(CONTRL YES YESQ))
          (0 (/NOT) 0 DLIST(CONTRL NO NOQ))
          ) AGAIN ))
```

Yes and no should be detected by using the TABLE mechanism. The tags YES and NO should be used. YESQ and NOQ will eventually be deleted. If yes or no is present in the input, and the appropriate tag YES or NO is not on the TABLE, the label AGAIN will cause a search for other keywords.

The scriptwriter may cause keywords to be added to the control script by using the function ADDKEY (see page A13). If this is done in the script called to start a given conversation (the "initialization script"), the keywords will effectively be a part of all scripts while they are being played. It is especially helpful to add substitutions for frequently misspelled words common to a particularly topic under discussion. For example:

```
ADDKEY(SA(0),'(SCALER=SCALAR.)),
```

The keywords added to the control script in this way remain only during the course of conversation with a single student and are not made a permanent part of the control script.

C. Student Control by Interruption

The control script contains not only commonly used keywords, but also some control keywords that allow the student to interrupt the conversation. These control keywords are added to every unit discussion script; therefore the student may use these control words at any time during the discussion. Depending on the type of control word used, the conversation may or may not return later to the point of interruption.

All the control keywords used have a rank greater than or equal to 10000, so that they will override an O'E on the TABLE (see the section on the TABLE machanism, page 23). The decomposition rule label for each keyword is AGAIN (see page 53). In brief, the present control words are the following:

"i understand" tells the computer to skip ahead to the following unit of discussion.

"i do not understand" causes a search for remedial material provided by the scriptwriter.

"go back" causes the computer to start the present unit of discussion again.

"quit" causes the discussion to be terminated immediately.

"when i say blt i mean bacon lettuce and tomato" causes the computer to record and remember for the rest of the conversation whatever substitution or abbreviation the student uses for his own convenience (for example, blt for bacon lettuce and tomato).

These control keywords and their synonyms are described more fully below.

Substitutions and Abbreviations

```
(I MEAN    (10000 IMEAN  (
           (0 SAY 0 I MEAN 0 DLIST(CONTRL IMEAN))
```

The student is able to make substitutions and abbreviations of the form "when i say X i mean Y". A substitution of the form (X = Y .) is added to the list of keywords of all scripts including the control script while they are being used in the discussion. The computer response to a statement of this type is "I UNDERSTAND. PLEASE CONTINUE."

Understanding

```
(AHA = I UNDERSTAND .)
(UNDERSTAND  (10000 UNSTAN  (
             (0 I 0 UNDERSTAND 0 DLIST(CONTRL UNSTAN))
(KNOW        (10000 KNOW  (
             (0 I 0 KNOW 0 DLIST(CONTRL UNSTAN))
```

This section enables recognition of a student's indication of understanding. It assumes he means that he understands the present unit of discussion. Since the present level of discussion is therefore finished, the system is sent to the label FINISH (see section F).

Lack of Understanding

```
(UNDERSTAND  (10000 UNSTAN  (
             (0 I 0 (/NOT) UNDERSTAND 0 DLIST(CONTRL NUNST))
(KNOW        (10000 KNOW  (
             (0 I 0 (/NOT) KNOW 0 DLIST(CONTRL NOTKN))
(REPHRASE=I NOT DLIST(/NOT) UNDERSTAND (10000(UNDERSTAND)))
```

These keywords should usually be used in connection with the TABLE (see page 22). When NUNST and/or NOTKN are placed as tags on TABLE, the computer goes to the appropriate label when the student indicates lack of understanding. If TABLE is not used and the student indicates lack of understanding, the computer types "TRY TO ANSWER ANYWAY."

Go Back

```
(GOBACK = GO BACK .)
(GO BACK   (10000 GOBACK   (
           (0 DLIST(CONTRL GOBACK))
```

When the student wants to begin the present unit of discussion again, the present script is read in again and the system is sent to the label START.

Quit

```
(QUIT     (10000  QUIT (
          (0 DLIST(CONTRL QUIT))
```

The student is able to quit the discussion and return to the CTSS command level.

D. Overriding the Control Script

It may be desired in some situations to ignore or override the whole keyword section of the control script that has been added to each script. Since the DLIST in the decomposition rule of every keyword contains the tag CONTRL (except those for yes and no), use of the tag CONTRL and a label as a pair in a list put on TABLE will send the system to the label if a control script decomposition rule fits the input. This effectively ignores the control script since the control mechanisms are overridden and not used. AGAIN would be a good label to use for this purpose. Individual keywords may be overridden by using the other tag in the DLIST (e.g., QUIT for the keyword quit) with a label as a pair in a list put on TABLE.

E. Unit Discussion Scripts

The method of input analysis used by ELIZA makes it relatively easy to program conversations that are linear in the sense that they follow a prescribed and predetermined line of argument. For long scripts this is undesirable, since one wants to adapt each conversation to the needs, desires, and depth of understanding of the individual student. The basic approach developed for use with the control script is to break the subject material into small units of discussion. The units are small and self-contained, so they may be ordered in any one of several ways. Other units may either follow or be used in the middle of a single unit. Many different branchings may be anticipated in a unit according to expected student responses to questions. With a number of units, each with multiple branchings, the number of paths the discussion may take becomes very large without the scriptwriter having to program each separate path in its entirety.

Each unit of discussion is contained in a single script. Within each unit, the context is limited to a certain topic, and a linear predetermined discussion of only three or four interchanges may be used. Thus, each unit script is relatively easy to program, without losing overall flexibility. Also, since each unit is independent, more units can easily be added to supplement existing units.

F. Connections Between Units, Sub-scripting

There are three ways that a given unit of discussion may be started. First, the end of the one unit may be reached in the course of conversation and the next appropriate unit determined. Then the new unit is called from disk storage and begun. Second, a new unit may be needed in the middle of another unit. Then the computer is instructed to remember where the original discussion stopped, and the new unit is called. Third, when the new unit is finished, the original disucssion unit is resumed. Since the discussion units are independent, they may be used either sequentially or as sub-units of one another.

It is helpful to introduce the terms level and sub-level to characterize the method of using discussion units. Units are on the same level if they follow sequentially in the discussion in the first manner described above. Units are on a sub-level and are called sub-scripts if they are used within another unit in the second manner described above. The two methods may be combined in a single discussion, and sub-levels of conversation used to any depth desired.

The technique developed for controlling the introduction of each unit of discussion can handle a wide

variety of structures of conversation. Particularly important is the fact that a discussion may continue on a sub-level through any number of units. When the conversation finally returns to the unit interrupted by going to the sub-level, it may return from a unit different from the first unit on the sub-level.

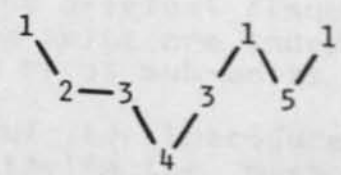
The following conversation illustrates the use of scripts as units of discussion. It is designed to show the structure of a conversation programmed in this manner. The diagrammatic representation of the conversation shows the order in which the scripts are called and the function of each. Note especially that FOUR SCRIPT is not itself concerned with the guessing of the word COUPLE, and might just as easily be used to quiz a student on the alphabet. This is the essence of independent unit scripts.

Example Conversation

```
r eliza
W 2021.4
WHICH SCRIPT PLEASE
one

I AM THINKING OF A WORD THAT MEANS TWO. WHAT DO YOU
THINK IT IS. QQ
pair

NO, THAT'S NOT IT.
THE WORD I AM THINKING OF HAS SIX LETTERS. TRY AGAIN.
double

WRONG AGAIN. HERE IS A HINT.
WHAT IS THE THIRD LETTER OF THE ALPHABET. QQ
c

RIGHT YOU ARE.
MY WORD BEGINS WITH THAT LETTER. TRY AGAIN.
couple

YOU ARE RIGHT. COUPLE IS THE WORD.

I AM THINKING OF A WORD THAT MEANS NEW. WHAT DO YOU
THINK IT IS. QQ
recent

RIGHT. RECENT IS THE WORD.
THIS IS THE END OF THIS CONVERSATION.
R 8.650+6.066
```

Diagram of the Conversation

(numbers stand for unit scripts)



ONE SCRIPT

```
(PROG       (ONE
            LABEL=POPTOP(DAHIN),
            GOTO(LABEL).
*START      STOUT='(CONCAT NWORD 9),
            STOUTN='(NOTYPE),
            OWN(PATH.),
            PATH='(TWO FIVE),
*LEAD       IF LEMPTY(PATH) THEN
              TYPE('THIS IS THE END OF THIS CONVERSATION. '),
              GOTO QUIT :
            TYPE('LINE(1) '),
            NEWTOP('(LEAD ONE),STORE),
            PLACE='START,
            SCRNAM=POPTOP(PATH),
            GOTO SUBSCR .
*FINISH     GROUP=0, NEWTOP('FINISH,DAHIN).
*CHANGE     GROUP=0, NEWTOP('CHANGE,DAHIN).
*SUBSCR     GROUP=0, NEWTOP('SUBSCR,DAHIN).
*QUIT       GROUP=0, NEWTOP('QUIT,DAHIN).
*AGAIN      KKK(KA(GROUP)).
*NOKEY      GROUP=0, NEWTOP('NOKEY,DAHIN).
            END)
```

TWO SCRIPT

```
(COUPLE   (COUPLE          (
          (0 DLIST(COUPLE)) () AGAIN
          ) AGAIN ))
(PROG     (TWO
          LABEL=POPTOP(DAHIN),
          GOTO(LABEL).
*START    TYPE('I AM THINKING OF A WORD THAT MEANS TWO.
           WHAT DO YOU THINK IT IS.QQ '),
          NEWTOP('(COUPLE TWOA O'E TWOB),TABLE),
*TWOA     TYPE('RIGHT. COUPLE IS THE WORD. '),
          GOTO FINISH .
*TWOB     TYPE('NO, THAT'S NOT IT. '),
          PLACE='START,
          SCRNAM='THREE,
          GOTO CHANGE .
*FINISH   GROUP=0, NEWTOP('FINISH,DAHIN).
*CHANGE   GROUP=0, NEWTOP('CHANGE,DAHIN).
*SUBSCR   GROUP=0, NEWTOP('SUBSCR,DAHIN).
*QUIT     GROUP=0, NEWTOP('QUIT,DAHIN).
*AGAIN    KKK(KA(GROUP)).
*NOKEY    GROUP=0, NEWTOP('NOKEY,DAHIN).
          END)
```

THREE SCRIPT

```
(COUPLE   (COUPLE          (
          (0 DLIST(COUPLE)) () AGAIN
          ) AGAIN ))
(PROG     (THREE
          LABEL=POPTOP(DAHIN),
          GOTO(LABEL).
*START    TYPE('THE WORD I AM THINKING OF HAS SIX
           LETTERS. TRY AGAIN. '),
          NEWTOP('(COUPLE THREEA O'E THREEB),TABLE).
*THREEA   TYPE('YOU ARE RIGHT. COUPLE IS THE WORD. '),
          GOTO FINISH .
*THREEB   TYPE('WRONG AGAIN. HERE IS A HINT. '),
          PLACE='START,
          SCRNAM='FOUR,
          NEWTOP('(THREEC THREE),STORE),
          GOTO SUBSCR .
*THREEC   TYPE('MY WORD BEGINS WITH THAT LETTER.
           TRY AGAIN. '),
          NEWTOP('(COUPLE THREEA O'E THREED),TABLE).
*THREED   TYPE('YOU HAVE HAD ENOUGH GUESSES.
           COUPLE IS THE WORD. '),
          GOTO FINISH .
*FINISH   GROUP=0, NEWTOP('FINISH,DAHIN).
*CHANGE   GROUP=0, NEWTOP('CHANGE,DAHIN).
*SUBSCR   GROUP=0, NEWTOP('SUBSCR,DAHIN).
*QUIT     GROUP=0, NEWTOP('QUIT,DAHIN).
*AGAIN    KKK(KA(GROUP)).
*NOKEY    GROUP=0, NEWTOP('NOKEY,DAHIN).
          END)
```

FOUR SCRIPT

```
(C          (C                  (
            (0 DLIST(C)) () AGAIN
            ) AGAIN ))
(PROG       (FOUR
            LABEL=POPTOP(DAHIN),
            GOTO(LABEL).
*START      TYPE('WHAT IS THE THIRD LETTER OF THE ALPHABET.QQ '),
            NEWTOP('(C FOURA O'E FOURB),TABLE).
*FOURA      TYPE('RIGHT YOU ARE. '),
            GOTO FINISH .
*FOURB      TYPE('NO. TRY AGAIN, THIS SHOULD BE EASY. '),
            GOTO START .
*FINISH     GROUP=0, NEWTOP('FINISH,DAHIN).
*CHANGE     GROUP=0, NEWTOP('CHANGE,DAHIN).
*SUBSCR     GROUP=0, NEWTOP('SUBSCR,DAHIN).
*QUIT       GROUP=0, NEWTOP('QUIT,DAHIN).
*AGAIN      KKK(KA(GROUP)).
*NOKEY      GROUP=0, NEWTOP('NOKEY,DAHIN).
            END)
```

FIVE SCRIPT

```
(RECENT   (RECENT          (
          (0 DLIST(RECENT)) () AGAIN
          ) AGAIN ))
(PROG     (FIVE
          LABEL=POPTOP(DAHIN),
          GOTO(LABEL).
*START    TYPE('I AM THINKING OF A WORD THAT MEANS NEW.
           WHAT DO YOU THINK IT IS.QQ '),
          NEWTOP('(RECENT FIVEA O'E FIVEB),TABLE).
*FIVEA    TYPE('RIGHT. RECENT IS THE WORD. '),
          GOTO FINISH .
*FIVEB    TYPE('NO, THAT'S NOT IT. '),
          PLACE='START,
          SCRNAM='SIX,
          GOTO CHANGE .
*FINISH   GROUP=0, NEWTOP('FINISH,DAHIN).
*CHANGE   GROUP=0, NEWTOP('CHANGE,DAHIN).
*SUBSCR   GROUP=0, NEWTOP('SUBSCR,DAHIN).
*QUIT     GROUP=0, NEWTOP('QUIT,DAHIN).
*AGAIN    KKK(KA(GROUP)).
*NOKEY    GROUP=0, NEWTOP('NOKEY,DAHIN).
          END)
```

In the programming of a unit script, there are three ways to call the next unit script.

(1) It may be called as the next script on the same level as the present script. To program this case, the scriptwriter must set values for two variables. SCRNAM must be set equal to the name of the next script, and PLACE must be set equal to the label in that script to which control will be transferred. Then, control must be transferred to the label CHANGE. For example, in TWO SCRIPT:

```
PLACE='START,
SCRNAM='THREE,
GOTO CHANGE .
```

This will cause THREE SCRIPT to be read in and control transferred to the label START.

(2) The next script may be called as a sub-script as an interruption to the present script. In order to remember the point of interruption, a list containing the label to which control should be transferred when the present script is returned to and the name of the present script must be put on the top of the list STORE. SCRNAM and PLACE must be set to appropriate values for the sub-script, and control must be transferred to the label SUBSCR. For example, in THREE SCRIPT:

```
NEWTOP('(THREED THREE),STORE),
PLACE='START,
SCRNAM='FOUR,
GOTO SUBSCR .
```

This will cause FOUR SCRIPT to be read in as a sub-script of THREE SCRIPT and control transferred to the label START. When that level of conversation is finished, control will be transferred to the label THREED in THREE SCRIPT.

(3) From a sub-level, the interrupted script of the next higher level is called back in. The only statement required in this case is to transfer control to the label FINISH, since the information regarding the point of return was stored in the list STORE when the sub-script level was started. For example, in FOUR SCRIPT:

```
GOTO FINISH .
```

This will cause the control script to read in the interrupted script and transfer control to the label specified as the location for return. In this example, that would be the label THREED in THREE SCRIPT.

In summary:

(1) Next script on same level--set SCRNAM and PLACE, go to CHANGE.

(2) Call a sub-script--put list containing label and script name for return on top of STORE, set SCRNAM and PLACE, go to SUBSCR.

(3) Reached end of discussion in a script--go to FINISH.

G. Techniques of Using Unit Scripts

Format

In order for the control script to function properly, the unit scripts must all have certain common labels and programming. To facilitate the writing of unit scripts, the necessary common program is provided in a script named UNIT SCRIPT. To write, for example, a unit script named ABLE SCRIPT, one begins with the commands indicated by the dashes (-) below:

```
- edl unit script
  W 1507.8
  Edit
- file able
  *
  R 2.833+.900
```

The other lines are computer responses. This produces a copy of UNIT SCRIPT filed under the new name ABLE, which can then be modified by editing (by EDL) to become a unit discussion script. To use the new copy, the scriptwriter must retype the first line to read:

```
(PROG      (ABLE
```

The keywords can be added above the PROG section, and the program for the script added beginning with the label START. UNIT SCRIPT is shown below.

UNIT SCRIPT

```
(PROG      (UNIT  TYPE('RETYPE PROG LINE ').
           LABEL=POPTOP(DAHIN),
           GOTO(LABEL).
*START
*FINISH    GROUP=0, NEWTOP('FINISH,DAHIN).
*CHANGE    GROUP=0, NEWTOP('CHANGE,DAHIN).
*SUBSCR    GROUP=0, NEWTOP('SUBSCR,DAHIN).
*QUIT      GROUP=0, NEWTOP('QUIT,DAHIN).
*AGAIN     KKK(KA(GROUP)).
*NOKEY     GROUP=0, NEWTOP('NOKEY,DAHIN).
           END)
```

Initialization

The first script (specified in response to WHICH SCRIPT PLEASE) should be used for initialization. Values should be set for STOUT and STOUTN (see page 58) in this script, and other variables, especially counters, should be set to initial values. The list of common variables should be specified by a COMMON statement (see page 20). Initial statements and questions could also be included. The value of doing all of this in the first script is that everything will be done for all the scripts that follow.

Led Discussions

Discussions may be led by the computer by specifying a list of scripts in the order to be used. A script must be used to control the conversation by calling each script in turn as a sub-script, and specifying an appropriate point of return. In the example discussion, ONE SCRIPT controls the conversation with the following program section.

```
        PATH='(TWO FIVE),
*LEAD   IF LEMPTY(PATH) THEN
          TYPE('THIS IS THE END OF THIS CONVERSATION. '),
          GOTO QUIT :
        TYPE('LINE(1) '),
        NEWTOP('(LEAD ONE),STORE),
        PLACE='START,
        SCRNAM=POPTOP(PATH),
        GOTO SUBSCR .
```

The list of scripts is usually named PATH since the computer determines the path of the conversation. Each script is called in turn by POPTOPing a script name off PATH. The point of return from the sub-script level is specified as the label LEAD, which will cause the whole process to be repeated. When PATH is empty, the end of the led conversation has been reached.

It is easy to have two or three possibilities for led discussions, and to give the student a choice among them by appropriate keywords and decomposition rules in the initialization script. The scriptwriter could also create a special script, whose name would be the last one on PATH, which would ask the student if he wanted to go on with another choice. If he does, PATH could be redefined, setting up another led discussion. This is one example of the use of the last script on PATH to set up another PATH.

Remembering the List on TABLE

When a label is used in the keyword section and there is a list on TABLE, the list on TABLE is taken off as part of the normal procedure (see the TABLE mechanism, page 22).

In the case of labels such as NOKEY, however, it is usually desired to put the list back on TABLE. After an input is processed, the name of the last list taken off TABLE is remembered as ELBAT and can be put back on again. In the unit scripts, the list is automatically put back on TABLE when the system goes to the labels NOKEY and AGAIN. If the scriptwriter wants to process an input without putting a list on TABLE, he should include the command LIST(ELBAT) for that input. Do not write MTLIST(ELBAT), since that would get rid of the labels.

Long Units

Long discussion units having more than three or four interchanges may be programmed by linking a number of small unit scripts together on the same level of discussion. This is done by setting PLACE and SCRNAM to appropriate values, and then GOTO CHANGE.

H. Other Mechanisms of Unit Scripts

By using UNIT SCRIPT (page 50) as a model for writing unit scripts, the scriptwriter has available certain useful mechanisms. The purposes of the labels CHANGE, FINISH, and SUBSCR have already been explained. The other mechanisms are described below.

*AGAIN

This section causes the system to reprocess the input, looking for the next highest ranking keyword. It is especially useful to use this label in the keyword section as a no decomposition rule label since this will cause the program to look at another keyword when no decomposition rules fit the input (see TWO SCRIPT on page 44).

*NOKEY

This uses the NOKEY mechanism in the control script, which randomly selects one of twelve phrasings of "PLEASE REWORD YOUR STATEMENT" each time the system goes to the label NOKEY. If any list is taken off TABLE in the processing of the input, it is put back on TABLE.

*QUIT

This section causes the script playing to terminate from the control script and returns control to the CTSS command level. This should be used instead of the function QUIT.

STOUTN

STOUTN is a list that controls the recording of the student part of the conversation (see page 58). STOUTN should be set equal to a list that includes the appropriate printing code words including NOTYPE (see page 55). STOUTN is intially set equal to (NOTYPE) by the control script.

TEM

There is a list named TEM (for temporary) that is created by the control script. It may be used as a list that is needed for temporary storage. Each time it is used for a different purpose it should be recreated by LIST(TEM).

LABEL

The variable LABEL is always equal to the last label POPTOPed off the the list DAHIN. It is sometimes useful when something has gone wrong and the scriptwriter wants to know where the system is.

Chapter 7 -- READING, WRITING, AND PRINTING

A. Definition of Terms

There are two areas where scripts or other files may be located. When a script is written (by using edl), it is stored as a file in the disk storage of the computer. When a script is being played, a copy is taken from the disk and put in the memory of the computer. The ELIZA system has control when a script is being used in memory. Reading refers to a transfer of a file copy from the disk to memory. Writing refers to a transfer from memory to a disk file. Printing refers to a transfer from memory to the console typewriter. Output refers to the characters transferred in either writing or printing.

B. Reading and Writing

In order to either read or write a file, the file must be opened. Only one file may be open at any one time for either reading or writing. Therefore, to read or write another file, the open file must be closed. The OPL functions that deal with reading, writing, and printing are described in the Appendix on page A22. The following section describes printing and writing in detail.

C. Printing and Writing

The two functions that control printing by the computer are TYPE and TXTPRT. Either one can also control writing onto disk files, for example recording all or part of a dialogue between student and computer. TYPE can also cause spaces, tabs, and carriage returns to be included in the output. For a complete description of its capabilities, see page A22.

Both TYPE and TXTPRT require instructions as to where and how to carry out the printing and writing. Printing can be done on the student console (with or without the parentheses that surround lists) and/or on a storage disk (in which case a name must be given for the file in which the recording is to be stored).

D. Output Format--The Code Words

The instructions to the computer are provided by an instruction list, which contains certain code words. The order of the code words on the list is not important, except for NWORD and DISK where the code consists of more than one word. The code words provide different options for printing and writing, and all of them do not have to be included on the instruction list. The code words are described below.

NWORD number

This code controls the maximum number of characters printed per line. NWORD must be followed immediately by an integer that is the number of words of 6 characters to be printed per line. For example, NWORD 8 sets the maximum line length at 48 characters per line. If this code word is not included, the system sets the number of words to be 14, or a maximum line length of 84 characters.

NOCR

When this code word is included, there will be no carriage return at the end of a line. It should only be used with printing that will not exceed one line.

NOTYPE

Printing will not occur on the console typewriter, but writing will still occur on the disk if specified.

CONCAT

The parentheses surrounding the list and the first level of sublist parentheses on a list are omitted in the output. For example, concatenated sentences are usually constructed by NEWBOTing the pieces onto a list, and use of this code word will cause printing and writing without extra parentheses appearing in the output. The list ((THIS)(IS)(AN EXAMPLE)) will be printed as THIS IS AN EXAMPLE.

LPRINT

Description lists are included in the output if they exist. Also, a set of parentheses is put around the output.

TAGS

When this code word is used, tag lists (e.g. (/FEMALE FAMILY)) are also included in the output.

DISK name1 name2

This code word causes the output to be written onto a disk file as well as printed on the console. The two words immediately following DISK specify the name of the file, as indicated by "name1 name2". For example, the words DISK SPACE TIME will result in the writing of a file named "SPACE TIME". If name2 is the word OUT, it will be changed to the user's programmer number. For example, DISK X OUT will result in the writing of a file named "X 9740" if the user's programmer number is 9740. This allows many users of the same script to operate at the same time. If the file already exists on the disk, the new output will be automatically appended to the file. DISK does not close the file.

CLOSE

Use of this code word closes the output file after the present output (see also the OPL function DSKCLS, page A24).

PROGPR

Use of this code word causes the output to be put into program format. This means that a carriage return and a tab will be inserted after each period, comma, and colon. Asterisks are interpreted as indicating labels, and will be preceded by a carriage return and followed by a tab.

E. Using TYPE and TXTPRT

In the TYPE function, the instruction list is not entered explicitly as part of the function. When the TYPE command is used, the system looks for the instruction list named STOUT (a contraction of "student output"). If no list named STOUT exists, the computer assumes that the command is "0" (zero), which means "on the typewriter". The TYPE function will print the outer pair of parentheses of a list unless the code word CONCAT is included on STOUT. (The TXTPRT function does not, see below.) Therefore it is wise to define STOUT to include the code word CONCAT as a minimum. The control script makes this definition. STOUT can, of course, be redefined in any script before or during the conversation.

In the TXTPRT function, the instruction list is indicated explicitly. The command

```
TXTPRT(L1,L2)
```

will result in the printing of the list L1 according to the code words contained in the list L2. Both lists may be written out in the function. The command

```
TXTPRT('(HELLO, MY FRIEND),0)
```

will result in the printout

```
HELLO, MY FRIEND
```

on the student typewriter. The instruction "0" (zero) in place of the list L2 is the simplest explicit instruction and means "on the typewriter". The TXTPRT function does not print the outer pair of parentheses that surround the list L1.

If this conversation were being recorded on the disk as well as being printed on the typewriter, the command might read

```
TXTPRT('(HELLO, MY FRIEND),STOUT)
```

where earlier in the program the list STOUT is defined, for instance, as

```
STOUT='(CONCAT CLOSE DISK XX YY)
```

In brief the command says "print the list (HELLO, MY FRIEND) on the typewriter without the outer and first level sub-list parentheses (CONCAT) and also append it to the disk file (DISK) whose first name is XX and whose second name is YY".

By using a single list such as STOUT in all TXTPRT commands, the entire conversation will be printed according to the code words on the list STOUT. If the scriptwriter wants to start or stop recording the conversation, only the code words on the list STOUT have to be changed. This would also affect all the TYPE commands.

F. Recording Conversations

In order to record conversations on disk files, both sides of the dialogue must be recorded. Both STOUT and another list STOUTN must be set to include the appropriate code words. For example:

```
STOUT='(CONCAT CLOSE DISK TRAIN OUT),
STOUTN='(NOTYPE CONCAT CLOSE DISK TRAIN OUT),
```

The computer side of the dialogue can then be recorded by using TYPE(L) or TXTPRT(L,STOUT), where L is the list containing the output. The student side of the dialogue is recorded automatically according to the code words on the list STOUTN. Thus, both sides of the dialogue will be recorded on a disk file named TRAIN progn, where "progn" will be the user's programmer number.

Chapter 8 -- WHEN THINGS GO WRONG

The scriptwriter must usually do two things when a script is not working properly. First, the source of the error must be located, not always an easy task. Second, the error must be corrected, which is sometimes easy, like a missing parenthesis, and sometimes difficult and may involve a lot of reprogramming. This section is concerned with ways of locating and correcting errors, and avoiding some of the more common ones.

A. Frequent Errors

Of all the possible errors that can be made in writing a script, the following are the most frequent. When looking for a mistake, it will be helpful to check for these first.

1. Omitting a parenthesis where it is needed. Keyword structures and certain functions such as TXTPRT are the most susceptible.
2. Omitting the END) at the end of the program.
3. Making incomplete or incorrect erasures during the editing of a script.

B. Tracing

The OPL function TRACE is helpful in locating the source of an error. TRACE has two modes. The normal mode is OFF. When the TRACE is turned ON by including TRACE(ON) as a command, the system will print the functions and arguments it executes. Much of the output may seem like gibberish, especially at first, but it should provide good clues as to what is going wrong. Understanding will come only through use. The TRACE may be turned OFF by including TRACE(OFF) as a command. The TRACE may also be turned ON and OFF by using $ as the first word of an input followed by the TRACE command (see page 10).

C. Testing

It is sometimes helpful to test small sections of an OPL program to see how they are functioning. In order to avoid writing a whole script, there is a script that will test these sections called EVAL SCRIPT (stored in m5347 cmf101 on Comp Center).

EVAL SCRIPT

```
(PROG       (EVAL
            GOTO(POPTOP(DAHIN)).
*START      TXTPRT('(PLEASE BEGIN),0).
*NOKEY      EVAL(INPUT), TXTPRT('(R),0).
            END)
```

EVAL SCRIPT executes the input as an OPL program and types R when it is finished. The small sections of program to be tested can be typed in as input for EVAL SCRIPT. They may also be tested by using $ as the first word of an input consisting of a small section of program (see page 10).

D. List of Rules

Trouble will result if the following rules are disobeyed.

1. Always close all opened parentheses.
2. Do not use more than six characters in labels, keycodes, script names, and variables. These also must start with a letter.
3. Make sure that the labels START and NOKEY are included in every script.
4. Do not put a period (.) immediately following a number--separate by a space (write 2 . rather than 2.).
5. Do not use the following symbols in a program or in an input--they will not be understood:
   & % ! ¢ ¬ | < > _ backspace
6. Include labels in all appropriate places in a keyword structure, even if they will never be used. For example, at the end of a keyword structure:
   (0) () LABEL
) NOKEY))
   The NOKEY will never be used but must be included.
7. Do not use anything outside of the keyword or program sections of a script; even a blank space at the end of a script causes problems.
8. Do not use the characters END) anywhere except to close off all parentheses. The command POPTOP(END), will cause problems.
9. Follow all commands (except IF and FOR) with a comma or a period.
10. Do not forget to close IF and FOR statements with a colon (:), and do not use the colon anywhere else in the program.
11. Do not use // for anything except comments--remember to close each comment.
12. Do not use more than five blank spaces together. If spaces are to be printed, use the TYPE function.
13. Do not read a script into the presently active group area.
14. Do not have two labels in the same script that are the same.
15. Do not use a precedence number higher than 262143.
16. Do not use more than one equals sign (=) in a command: X=Y=Z will not work.
17. Do not use .E. for "equals" in an assignment statement, and do not use = for "equals" in a Boolean expression.
18. Make sure all variables to be used are first specified in either OWNLIST or COMMON.
19. Make sure a period follows the last variable in OWN and OWNLIST statements. This is not true for COMMON statements.

E. Special Names

The following names are reserved for use by the system or the control script, and should be used by the scriptwriter only in the ways described in this manual. Their proper usage is described on the indicated pages.

INPUT	3
DECOMP	4
SEMBLY	5
TABLE	22
ELBAT	23
O'E	23
KEY	28
EXP	28
DAHIN	18
SA	19
KA	19
GROUP	19
SCRPN	19
PROG	6
START	7
NOKEY	7
STOUT	57
STOUTN	58
STORE	48
LABEL	53
PLACE	48
TEM	53
SCRNAM	48
END)	6

The following words should not be used by the scriptwriter as variable names.

E'R
ABORT
RETURN
REPEAT
WHILE
WHERE
SAV

In addition, it is dangerous to use function names as anything (such as labels or variables) other than functions.

Chapter 9 -- THE SLIDE-DICTIONARY SYSTEM

by Michael J. Knudsen

November, 1967

The SLIDES system was developed as a Special Problem in Electrical Engineering by Michael J. Knudsen at the M.I.T. Education Research Center, in conjunction with Professor Joseph Weizenbaum, Dr. Merton J. Kahne, Dr. Judah Schwartz, and Dr. Walter D. Daniels.

Table of Contents

A. Introduction

The SLIDES system is intended to enable ELIZA users to write scripts which use in the instruction not only their own printed words but also pictorial, diagrammatic, or graphical information presented on photographic slides. A slide projector is commanded by the script to show the student any of a large number of slides stored in the projector's "carousel"-type magazine. Through an "X-Y" coordinate system similar to that used on road maps, the script can "point out" given objects or features of discussion on the slide by referring to that object's coordinates. Likewise the student may answer or ask questions of the script by typing coordinates of the point in question. Considering that most commercial teaching machines have some sort of built-in slide projection system, it is only natural that one of the most powerful teaching "machines," ELIZA, should also be able to utilize pictorial information.

To simplify matters for the scriptwriter, the scripts may be made independent of any particular set of slides, to the extent that a script may be written without knowing the actual coordinates of the significant features on the slides to be used with that script. In fact, the script may be written before the actual slides have been prepared. When writing the script, the author need only have in mind a fairly specific plan as to what objects, in what relations, the script will need to have available on slides. For example, in developing a tutorial on cell division, the writer might say "I assume I can obtain slides showing a cell in the successive phases of mitosis, of sufficient quality to show clearly the parts of the cell referred to in my scripts..." After writing the script, he may then go about securing the needed microphotos.

Once a particular set of slides has been made up for a script, some kind of "dictionary" must be supplied with this set, for translating names of objects referred to in the script into their exact location-coordinates on slides, and vice-versa. The script may need to refer the student to some object by giving a slide and the object's coordinates on it, but since the script was prepared separately from the slides, no coordinate information is built into the script; hence the need for the "dictionary" or auxiliary memory in which the script can "look up" the slide and coordinates of the object, and then type these out for the student or otherwise use them. Conversely, the script may ask the student to identify something by pointing it out (typing in its coordinates); then the dictionary must be consulted to see whether the student's coordinates do point to the correct answer.

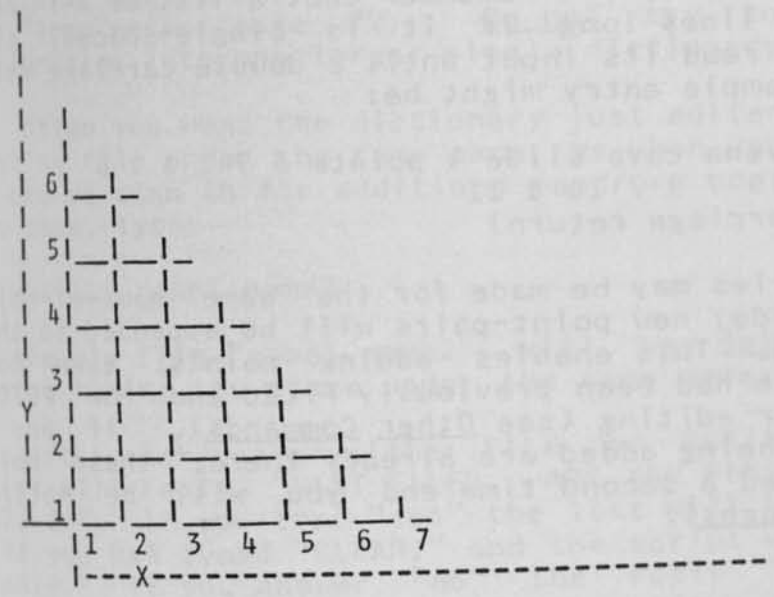
This chapter explains the use of the SLIDES system, which supplies the needed dictionary. There are two phases in working with a slide dictionary--first, given the particular set of slides for use with the script, a disk file containing the dictionary list is created, this process being called "editing." Editing is handled by a special script provided. The other phase, called "retrieval," involves the use of this dictionary list by the teaching script (or set of scripts). Retrievals, or "lookups," are performed by function calls in the teaching script calling the retrieval functions which have been defined for this purpose in the OPL language. These functions are all contained in a library file called "SLIDES LIBE," which is used exactly like "DEFINE LIBE"; see page A29. You may print out the SLIDES LIBE to study the operation of the functions.

(At present, a script must type out the number of the next slide to be used, and the student must select this slide by manual control. Also, the script must type the coordinates of a point in order to "point out" something to the student, who in turn must type in coordinates when referring to some point on the slide. The selection of slides will someday be automated by direct control of the projector carousel from the data phone, and we expect to develop a mechanical pointer attachment for the viewing screen, which, controlled in a similar manner, will point out objects directly and can be manipulated by the student to transmit coordinates back to the computer system. Thus the student will be freed from thinking about coordinate numbers. Automatic slide projection would also make it feasible to present large blocks of textual material on slides instead of typing them out.)

While everything described in this chapter works as described, the SLIDES system should not be regarded as a final, finished product. Like ELIZA, SLIDES is an evolutionary system, whose continued improvement depends on the "feedback" from the users. New functions are constantly being developed. We are anxious to hear any comments regarding bugs, possible new functions, changes to existing functions, questions on the use of the system, and comments on this chapter. All questions and comments should be addressed to Michael Knudsen, Education Research Center, Room 20-C-120, MIT, Extension 5383.

B. Editing

The coordinates of a point on the slide are determined by reference to the grid below, superimposed on the slide (this grid is presently engraved on the ground-glass screen in front of the student on which the slides are projected; it is also possible to have the grid physically printed on each slide.)



To edit a slide, use the terminal equipped with the projection system, i.e., the terminal to be used by students playing your script with the slides. Load the set of slides to be edited into the projector in any desired order; from now on, the number of each slide is its numbered position on the remote selector dial of the projector. Start the ELIZA system (if possible use "SYS", without the control script) and play the slide-editing script, EDITOR. Use manual control to select the slide to be edited. With the slide showing on the screen, for each object that you wish your script to recognize as being on that slide, note which squares (points) of the grid that object covers and make a list of these. Usually an object will cover several squares. If a square lies on the boundary of an object so that the object only partly fills the square, it is better to include that square in the list because of the nature of some of the retrieval functions, such as CENTER and OUTLIN. This is your decision, however.

For each object, type in its point-list for that slide in this format:

name-of-object SLIDE number POINTS x1 y1 x2 y2 x3 y3...

where "name-of-object" can be a phrase of any length and number of words, "number" is the number of this slide, and the X-Y pairs are integer numbers from the coordinate grid, all these being the list of squares covered by the object. The words SLIDE and POINTS must be inserted as shown, though not in capitals; these serve as delimiters for the decomposition rule in the editing script. At least one entry operation must be made for each object used by the script on that slide, but remember that a list of X-Y pairs may be several lines long if it is single-spaced, since ELIZA does not read its input until a double carriage return is given. A sample entry might be:

inferior vena cava slide 4 points 3 7 3 8 4 8
5 8 5 9 6 9 7 9 7 10 8 11
(double carriage return)

Additional entries may be made for the same name-of-object on the same slide; new point-pairs will be appended to those already entered. This enables adding points, even to a dictionary which had been previously FILEd and now FETCHed back for further editing (see Other Commands). If any of the new points being added are already there, these points will not be added a second time and you will be notified (see Error Messages).

Deleting

For various reasons, you may wish to remove some squares from the lists for some objects. To do this, type:

DELETE name-of-object SLIDE number POINTS x1 y1 ...

where everything is the same as for an entry, except that the list of X-Y pairs contains only those points to be deleted from that name on that slide.

To remove an object entirely from a slide, without typing all its points on that slide, use:

DELETE name-of-object SLIDE number

To eliminate a name from the entire set of slides, type:

DELETE name-of-object

Filing

When finished editing the entire set of slides for that dictionary, to file the list permanently in your disk tracks

type:

FILE name1 name2

where "name1 name2" is the desired name of the new file in your file directory. If there is already a file by that name, the new dictionary will be appended onto it; for ordinary purposes this is not recommended. (This appending process can be used to store several independent dictionaries on a single disk file, thus saving tracks; a particular dictionary can be found and read in by a special read "loop"--see page 72. Do not try to combine two dictionaries into one larger single dictionary this way!!)

Often you want the dictionary just edited to replace an existing file under the same name, as when you have FETCHed the old version in for additions and/or corrections. In this case, type:

REFILE name1 name2

and the old file "name1 name2" will be deleted, the new version taking its place under the same name.

After each use of either FILE or REFILE, the Editor script will reply, "LIST FILED. ARE YOU FINISHED WITH THIS LIST, QQ". If you type "yes" the list will be cleared just as if you had typed "CLEAR," and the script will say "LIST CLEARED." If you answer "no" the reply "LIST READY TO EDIT." will be given, and you may continue to add to the list. This is useful in building up a new dictionary in stages, REFILEing every so often to protect your work so far in case of a system "crash."

Other Commands

To aid in manipulating the dictionary lists, these other commands are recognized by the script:

CLEAR

Erases the dictionary list that was being edited, in preparation for working on another. Typing CLEAR has the same effect as quitting and playing the Editor script over again. Several files can be created without the need for quitting. Don't forget to FILE the list just finished before CLEARing!

FETCH name1 name2

This command reads the dictionary list "name1 name2" into the editing bay. (The "editing bay" is the list, called NAMES, in the editing script where the dictionary is built up.) Normally you would first give a CLEAR command unless you have just started the editor script. If the editing bay is not empty, the FETCHed list will be appended onto the one already in the bay. This can be used to combine two dictionaries, provided they have no names of objects in common (otherwise the retrieval functions cannot find anything under the second occurrence of any duplicated names). If the requested file is not found, the familiar "GOOF ON READING FILE" will be printed. If all goes well, Editor will type "LIST READY TO EDIT."

LIST

By itself, this will print (using TXTPRT) the entire contents of the editing bay, i.e., the present form of the dictionary, in straight linear form. Good for overall checks and seeing the structure of a dictionary list. However, if the dictionary is already rather long and you want only to check entries under a particular name-of-object, use:

LIST name-of-object

This lists the entries under the given name, one slide at a time. If nothing has been entered under the name (or you misspelled it), a message is printed.

QUIT

This request returns you from the ELIZA system to CTSS command level, terminating the editing. Remember to FILE your work before QUITing!

Error Messages

The Editor script is provided with checks to protect itself and the user against errors. Error responses may result from mistakes either in entries and deletions, or in using the additional commands just described. Messages involving entry and delete operations are:

"PLEASE TRY AGAIN."

Your request did not fit any recognizable format.

"POINTS NOT IN PAIRS. NO ACTION TAKEN."

Your list of X-Y points to be entered or deleted had an odd number of numbers. You probably omitted an X, Y, or space between two numbers. Because this error "scrambles" the points, the script checks for this before acting on the points--hence you must repeat the entire entry or deletion. An even number of such errors in one command cannot be detected.

"ALL POINTS ENTERED, EXCEPT THESE DUPLICATES--
...(list of points)... "

This indicates that the points listed after the message were already in the dictionary or appeared twice in this entry. All other points in your entry will have been entered as usual, so no harm is done. However, your apparent duplication may be due to a typographical error, so check your work.

"ALL POINTS DELETED, EXCEPT THESE NOT FOUND--
...(list of points)... "

Your list of points to be deleted contained some points not in the dictionary under the given name-of-object on the given slide. All other points in your command have been deleted as usual, but again, check your work.

"NO name ON SLIDE number "

You tried to delete a name entirely from a slide on which it was not listed as appearing. Check your spelling.

"NAME NOT FOUND--name "

You tried to delete a name completely which wasn't there--spelling?

C. Retrieval by Scripts

The functions to be described permit a script to make use of the name-coordinates information stored in dictionary lists. Before discussing the functions themselves, some preliminaries must be explained.

Internal Dictionary Lists

For use by a script, a dictionary list must be read from the disk file into a list in the ELIZA system. The list which receives the dictionary may be any list which has been created in the usual way. The retrieval functions will then operate on this list.

Reading-In Dictionaries

The desired dictionary is read into the desired list by using the OPL "DSKLST" function (see page A23). A recommended format, in two lines, is:

```
DSKLST(Name1,Name2,Dict),
DSKCLS(0,0,0),
  (rest of program)
```

where "Name1 Name2" is the name of the dictionary file, and "Dict" is the name of the list into which the dictionary is read. The zeros in DSKCLS are required dummy arguments.

The dictionary list should be declared COMMON so that all the scripts in your set can refer to it.

If several dictionary files have been concatenated, each successive call to DSKLST without calling DSKCLS in between will read the next list after the last one. After the last list has been read from the file, however, DSKLST will take on the function value 'DONE. Therefore you can construct a program "loop" to read in, say, the third list in such a combined file. Since each successive list is appended to the contents of the "Dict" list in your script, you must do a LIST(Dict) between repetions of DSKLST (but not DSKCLS, which must be done after the final desired list is read in. Note that if the 'DONE condition occurs, the file is automatically closed. To read in the third list:

```
FOR C=1 STEP 1 UNTIL C .G. 3 .OR. X .E. 'DONE DO
  LIST(Dict), X=DSKLST(Name1,Name2,Dict) :
DSKCLS(0,0,0),
  (rest of program)
```

There is no restriction on the number of separate dictionary lists that can be present at one time (each dictionary in a separate list, of course), so one script may refer to several dictionaries, since the retrieval functions are "told" which list to refer to. Unfortunately, the present memory space limitations do not encourage multiple dictionaries.

Loading the Functions

The retrieval functions must be defined, or loaded into the system so as to be known to the ELIZA evaluator. The functions are loaded from the SLIDES LIBE disk file by:

```
EVAL(LOAD('SLIDES,'( fn1 fn2 fn3.....))),
```

where "fn1 fn2 fn3 ..." need contain only those functions used in your script set, or required by those you use. Many of the SLIDES functions depend on others, as shown below:

Function	Requires these functions
NAM	VALIST
SLD	FIXIT
NAMPTS	VALIST
NAMSLD	VALIST, FIXIT
ATPT	FIXIT
CENTER	VALIST, FIXIT
OUTLIN	VALIST, FIXIT
MAXNAM	VALIST
MAXPTS	NAMSLD, VALIST, FIXIT
MAXSLD	FIXIT
MSLPTS	NAMSLD, VALIST, FIXIT
ISITIN	none
VALIST	none
FLOAT	none
FIXIT	none
VERT	OUTLIN, VALIST, FIXIT
HOR	OUTLIN, VALIST, FIXIT

The Retrieval Functions

Conventions

The functions are listed below with their call-argument formats. These functions all have certain call arguments in common. DICT is the name of the list into which the dictionary in which you want to "look up" the data has been read. OUT is the name of whatever list you want the output of the function to be loaded into. The output, if any, is NEWBOTed onto the present contents, if any, of OUT. DICT and OUT, in that order, are always the last two call arguments of the functions, except for the "MAXNAM" function, which does not use OUT, and ISITIN, a Boolean (true or false) function that takes any two lists as arguments. Of course, the names DICT and OUT are merely mnemonic--use any names you like. Other dummy variables shown as call arguments are:

Variable	Mode	Meaning
NAME	List name	Name of an object on a slide (must be a list containing the name and nothing else.)
SLIDE	Numeric--(Floating-point or integer)	Number of a slide
X	Numeric	Horizontal coordinate of a point
Y	Numeric	Vertical coordinate of a point

Function Descriptions

In the following descriptions, a typical question that each function "asks" of the dictionary is given. Also is shown the format of the functions output in the OUT list. Note that whenever NAME, SLIDE, X, or Y appear in an argument list, they are always input (given) data; output results appear in the OUT list and/or as the function's value. The normal return value of most of the functions is just the name of the list OUT.

When any retrieval function can find nothing in the dictionary to satisfy its input conditions and must return empty-handed, it returns the literal word 'NONE instead of the name of the list OUT, to which nothing has been added. Whenever a call to a retrieval function involves student input, your script should always check the results for the null case, as by "IF SLD(SLIDE,DICT,OUT) .E. 'NONE THEN GOTO NULL :", etc.

The Functions

NAM(NAME,DICT,OUT) frv: OUT or 'NONE

"List all the slides that show a NAME."

OUT=(SLIDE1 SLIDE2 SLIDE3 ...)

NAMPTS(NAME,DICT,OUT) frv: OUT or 'NONE

"List all the slides, together with the points on that slide, which show a NAME."

OUT=(SLIDE1 (X11 Y11 X12 Y12..) SLIDE2 (X21 Y21 X22 Y22..)..)

Note that OUT becomes a list of pairs, with slide numbers as attributes and as values those points on that slide that show a NAME.

SLD(SLIDE,DICT,OUT) frv: OUT or 'NONE

"What names appear on this SLIDE?"

OUT=(NAME1 NAME2 NAME3 ...)

The "inverse" function of NAM. Note that each NAME in OUT is itself a list, so OUT is a list of lists.

NAMSLD(NAME,SLIDE,DICT,OUT) frv: OUT or 'NONE

"What points on this SLIDE show this NAME?"

OUT=(X1 Y1 X2 Y2 X3 Y3 ...)

OUT is a list of data pairs which should be treated as such by your script. This function is the "intersection" of NAM and SLD.

ATPT(SLIDE,X,Y,DICT,OUT) frv: OUT or 'NONE

"What object NAMEs are at the point X Y on this SLIDE?"

OUT=(NAME1 NAME2 NAME3 ...)

Same output format as SLD. If more than one NAME appears in OUT, then all these objects overlap at the given point.

MAXNAM(NAME,DICT) frv: SLIDE or 'NONE

"Which slide has the largest (or most) NAME?"

OUT is not used.

The first of four "maximum" functions, MAXNAM returns the slide having the largest number of points listed under NAME. Note that the function itself takes on the return value.

MAXPTS(NAME,DICT,OUT) frv: SLIDE or 'NONE

"Which slide has the largest or most NAME, and on what points?"

OUT=(X1 Y1 X2 Y2 X3 Y3 ...)

This one does MAXNAM, plus it returns the "winning" set of points. Note the slide number is the function value, as with MAXNAM, but OUT is also used, with same format as for NAMSLD. If the NAME appears on no slides, MAXPTS takes on the value 'NONE and OUT is empty.

MAXSLD(SLIDE,DICT,OUT) frv: OUT or 'NONE

"What object covers the most points on this SLIDE?"

OUT=NAME

The inverse of MAXNAM, but unlike it, MAXSLD must use OUT because the single name is a list.

MSLPTS(SLIDE,DICT,OUT) frv: OUT or 'NONE

"Do MAXSLD and list the "winning" NAMEs points."

OUT=(NAME X1 Y1 X2 Y2 X3 Y3 ...)

The inverse of MAXPTS. Note the mixed contents of OUT--a list on top of a list of data pairs. You can POPTOP the name-list off to get at the point pairs.

CENTER(NAME,SLIDE,DICT,OUT) frv: OUT or 'NONE

"What point is the center of the NAME on this SLIDE?"

OUT=(X Y)

CENTER takes the arithmetic mean of all points on this SLIDE under this NAME to find the centroid of this object. This

may be useful when it is necessary to use a single point to point out a large object to the student. Scriptwriters should beware of crescent-shaped and other odd objects whose centroid may be outside the object--see the next function.

OUTLIN(NAME,SLIDE,DICT,OUT) frv: OUT or 'NONE

"Outline the NAME on this SLIDE."

OUT=(XT YT XR YR XB YB XL YL)

T, R, B, and L stand for top, right, bottom, and left. OUTLIN finds the farthest-out points in all four directions, and lists them in clockwise order, starting from 12:00. Even on a slide filled with irregular, overlapping large objects, CENTER and OUTLIN together can nail down one item fairly reliably. These two functions are expected to be useful with automatic pointer mechanisms also.

If CENTER or OUTLIN is used on a name that refers to more than one separate object on the given slide, the results require special interpretation. OUTLIN would return the topmost point of the highest such object on the slide, but the lowest point of the lowest such object, etc. In a cluster of several small objects of the same name, OUTLIN will find the extreme members of the group and thus delimit the cluster; CENTER will point to the density-weighted center of the cluster, which might not be any individual object.

ISITIN(L1,L2) frv: B

"Does the list L1 match contents with any list on L2?"

This is a Boolean function, taking the values 1 (true) or 0 (false). It does not refer to the dictionary but is provided for checking a list of names L2 (such as SLD or ATPT would return) for the specific name in L1. However it is used, every element of L2 must be a list.

Utility Functions: These are used by most of the retrieval functions, but are included here for your use if needed.

VALIST(L1,L2) frv: D or 'NONE

"Find the value of the attribute-list L1 on the pairs-list L2."

This function is identical to the OPL "VAL" function, except that the attribute whose value is sought is a list, not a datum. Normally the attributes are the object-names in the dictionary.

FLOAT(N) frv: F

A utility function that converts an integer to its floating-point equivalent.

FIXIT(N or F) frv: N

A utility function which accepts a number of either integer or floating-point mode, and returns an integer (the OPL "INTGER" function accepts only floating-point numbers). The retrieval functions use this to be able to accept SLDNO, X, and Y in either "numeric" mode.

Future Functions

In the near future we expect to develop high-level functions which test for certain spatial relations between objects on a slide, i.e., whether A is above, below, left of, inside, or outside B. The student might ask to talk about the cell on slide 16, but there are two cells on this slide, so the script asks him which one. The student could then answer "The one on the left."

The Editor script and the retrieval functions might also be modified to allow object-names to be "tagged" with various descriptions, modifiers, or subscripts, using the OPL DLIST feature. These tags could keep several objects with the same name on the same slide logically separate; conversely, the same tag on different names could mark these as all belonging to the same class. The scriptwriter could tag a particular cell on some slide as being the best all-around example of a cell, so if the student said "Show me a cell," that one would be selected.

Experimental Functions

Two spatial-relation functions of the type mentioned under "Future Functions" are available in tentative form:

VERT(NAME1,NAME2,SLDNO,DICT) frv: F or 'NONE

"What is the vertical relation of NAME1 to NAME2?"

If either name cannot be found, VERT returns 'NONE. Otherwise, ignoring the horizontal relation of NAME1 and NAME2, VERT returns the floating-point integer F with values: 2.0 if all points of NAME1 are above all points of NAME2 ("completely above"); 1.0 if some points of NAME1 are above all points of NAME2 and no points of NAME1 are below any of NAME2 ("above but overlapping"); -2.0 if NAME1 is completely below NAME2; -1.0 if NAME1 is below but overlaps NAME2; 0 if none of the above. Note that exchanging NAME1 and NAME2 for a given pair of objects just reverses the sign of the result.

HOR(NAME1,NAME2,SLDNO,DICT) frv: F or 'NONE

"What is the horizontal relation of NAME1 to NAME2?"

HOR is identical to VERT except that X-coordinates are checked. Substitute "to the right of" and "to the left of" for "above" and "below" in the description of VERT.

If either NAME1 or NAME2 in the above functions refers to more than one separate object on the slide, the frv's have special meanings--see the notes under OUTLIN, page 77. (Both HOR and VERT work through OUTLIN.)

While the author welcomes comments on any aspect of the SLIDES system, comments on the experimental functions are especially important in developing a good set of spatial-relations functions. Is the resolution of VERT and HOR (number of cases recognized, now 5) sufficient? Might you want to compare objects on two different slides? Please address all suggestions to Michael Knudsen, Education Research Center, Room 20-C-120, MIT.

D. Appendix

Dictionary-List Structure

A dictionary list is a list structure three levels deep--the main list, its sublists, and sublists of those sublists. The top list, the name list, consists entirely of sublists, arranged in attribute-value pairs. Each attribute is the name of some object, and its value is a second-level slide list. A slide list is also an attribute-value set. Here the attributes are slide numbers (each a datum), whose values are the third-level point lists. Point lists contain pairs of X and Y coordinate numbers (all data, no more sublists).

To find a picture of some object, we would first find its name in the name list. In its value, the slide list immediately following it, we would have a set of slides to choose from. After each slide number, the point list for that number would tell which points on that slide were covered by the object. Note that, in the entire structure, each object name will appear only once, but a slide number will appear once on the slide list of every object-name listed as appearing on that slide.

As list structures go, this one is relatively simple. An understanding of it should enable anyone familiar with the basic OPL functions to understand the workings of the editor script and the retrieval functions.

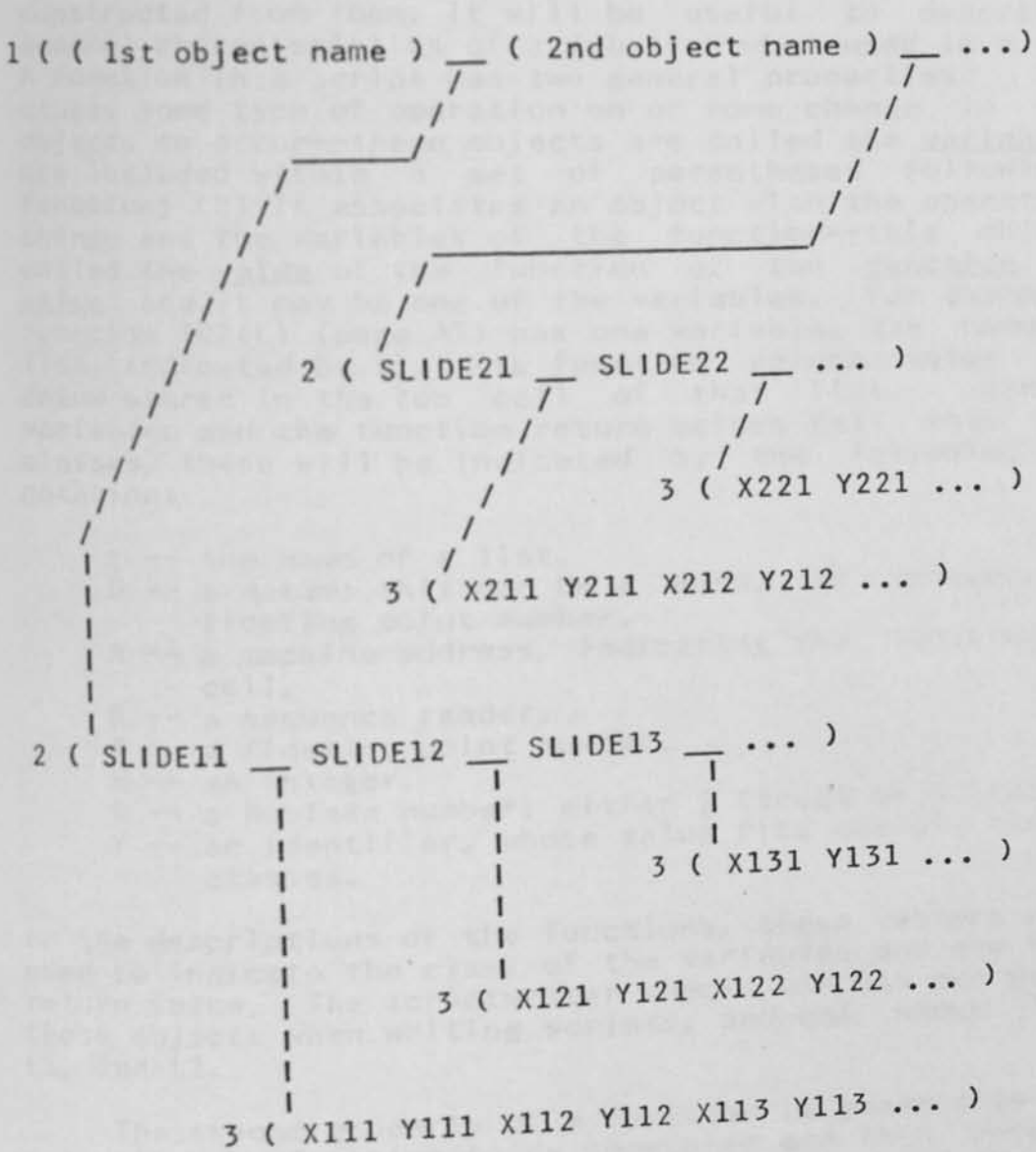
A diagram of the dictionary list structure is given on the next page.

Demonstration Script for Retrieval Functions

You may try out the retrieval functions by playing the test script, SLIDES (uses SLIDES LIBE), which accepts questions for each function (similar to the "questions" given in the function descriptions) and prints out the results for whatever dictionary file you have "FETCHed." This script may also be of value in checking large dictionary files. The appropriate question formats can be seen by printing this script and looking at the key structure. Note that there are several printout routines in this script for the various formats in which different functions deliver their output.

Diagram of Dictionary List

The name list is marked with a 1, slide lists by 2, and point lists by 3. These numerals are not stored in the structure at all, but are just labels for this diagram.



THE OPL FUNCTIONS

General Characteristics of Functions

A function is an instruction or command that tells the computer to do something. Before listing and describing the OPL functions and some of the operations that can be constructed from them, it will be useful to describe the general characteristics of a function when used in a script. A function in a script has two general properties: (1) it causes some type of operation on or some change in certain objects to occur--these objects are called the variables and are included within a set of parentheses following the function; (2) it associates an object with the operation or change and the variables of the function--this object is called the value of the function or the function return value, and it may be one of the variables. For example, the function TOP(L) (page A5) has one variable, the name of a list (indicated by L). Its function return value is the datum stored in the top cell of that list. Since the variables and the function return values fall into certain classes, these will be indicated by the following letter notation:

L -- the name of a list.
D -- a datum; this can be a word, an integer, or a floating point number.
A -- a machine address, indicating the location of a cell.
R -- a sequence reader.
F -- a floating point number.
N -- an integer.
B -- a Boolean number; either 1 (true) or 0 (false).
Y -- an identifier, whose value fits one of the above classes.

In the descriptions of the functions, these letters will be used to indicate the class of the variables and the function return value. The scriptwriter should use his own names for these objects when writing scripts, and not names like D, L1, and L2.

The second property of a function is especially useful for using the function to do something and then becoming a variable of another function. In this use, the name of the first function replaces a variable of the second function. An example should make this clear. The function LIST(L) (page A3) is a function of one variable L. Its properties are: (1) it creates a new list L; (2) it gives the name of the list, which is L, as its function return value. The function NEWTOP(D or L1, L2) (page A3) is a function of two variables, one of which is either D or L1, the other is L2. This function puts the datum D or the list L1, whichever

category the variable fits, on the top of the list L2. To show how a function return value can be used as a variable, examine the statement:

```
NEWTOP('WORD,LIST(STACK))
```

Since all operations are performed from the inside of the parentheses working out, the function LIST is performed first, creating the list named STACK, disposing of any old list named STACK, and putting the function return value, which is STACK, in place of LIST(STACK) in the expression. STACK is now empty and the statement looks to the computer like:

```
NEWTOP('WORD,STACK)
```

The function NEWTOP is performed next, which places the datum WORD on top of the list named STACK. NEWTOP itself has a function return value, but it is not important for this example. STACK finally looks like: (WORD) .

In identifying the functions in this appendix, the following format will be used:

```
FUNCTION(V1,V2,...,Vn)          frv: VALUE
```

where the "Vs" are the variables and "frv" stands for "function return value". In a script, the function return value may be identified by using an assignment statement:

```
X=FUNCTION(V1,V2,...,Vn)
```

The value of the identifier X will then be the function return value of FUNCTION.

LIST FUNCTIONS

This section contains those functions that are primarily concerned with operations on and information retrieval from lists.

LIST(L) frv: L

This function initializes a list by creating an empty list whose name is L, which is also the function return value. If L already exists, it becomes an empty list.

MTLIST(L) frv: L

This function empties the list L, i.e., its cells are returned to available space. The value of the function is the name of the now empty list. This function does not empty or remove any description lists which are attached to the list L.

LEMPTY(L) frv: B

This function returns the Boolean value 1 if the list L is empty; otherwise, this function returns the Boolean value 0. The list L is not changed.

COUNT(L) frv: N

This function returns the number N of elements or cells in the list L. The list L is not changed.

LISTOF(Y1,Y2,...,Yn) frv: L

This function takes the data Y1,Y2,...,Yn and puts them in that order as the elements of a list whose name L is the function return value. Each Y can be either a datum or the name of a list.

NEWTOP(D or L1, L2) frv: A

This function "places" the datum D or the list L1 in the top cell of the list L2. The value of the function is A, the machine address of the added datum.

NEWBOT(D or L1, L2) frv: A

This function "places" the datum D or the list L1 in the bottom cell of the list L2. The value of the function is A, the machine address of the added datum.

INLSTR(L1, A or L2) frv: L1

This function "inserts" the list L1 on the top of the list L2 or to the right of the machine address A. The list L1 is emptied by this function, and the name L1 is returned as the function value.

INLSTL(L1, A or L2) frv: L1

This function "inserts" the list L1 on the bottom of the list L2 or to the left of the machine address A. The list L1 is emptied by this function, and the name L1 is returned as the function value.

Note: There is a difference between "placing" a list L1 in some location in a list L2 and "inserting" a list L1 in some location in a list L2, as can be seen by comparing the functions NEWTOP and INLSTR. When a list is "placed" in a location, only the name of the list is placed there, and the list becomes a sublist (see page 31). When a list is "inserted" in a location, the contents of the list are actually placed in that location, so that the result is a single list.

Note: The pair of functions LINLST and STRLST involve the difference between a linearized list and a sublisted structured list. A list may contain references to other lists (i.e. sublists) when such sublists are "placed" on the original list, as the NEWTOP function does. As mentioned before, only the name of the sublist (with an indicator to show that it is the name of a sublist) is put on the list. A linearized list has had the contents of all sublists filled in between parentheses (as literal characters) on the list. The difference between these two states of a list cannot be detected by the TXTPRT function, since this causes a sublisted structured list to be printed by filling in the contents of the sublists between parentheses in the printed output. However, the difference can be detected by the MATCH function (page A14), since the two states of a list are different in structure.

LINLST(L1,L2) frv: L2

This function takes the list L1, makes a linearized copy of it, and "inserts" it on the bottom of the list L2. The list L1 is not changed by this function.

STRLST(L1,L2) frv: L2

This function takes the list L1, makes a structured copy of it, and "inserts" it on the bottom of the list L2. The list L1 is not changed by this function.

LSLCPY(L1,L2) frv: L2

This function makes a copy of the list L1 and "inserts" it on the bottom of the list L2. If L1 has sublists, only the name of the sublist is copied. The list L1 is not changed by this function.

LSSCPY(L1,L2) frv: L2

This function makes a copy of the list L1 and "inserts" it on the bottom of the list L2. If L1 has sublists, all of its sublists and their sublists are copied with their new names replacing the old names in the copy. The list L1 is not changed by this function. Since this function copies both lists and sublists, which is very time and space consuming, LSLCPY should be used to copy lists instead of LSSCPY except in very special cases.

NULSTR(L1,A,L2) frv: L2

This function causes the list L1 to be split into two separate lists at A, the machine address of a cell on the list L1. A new list L2 is created that contains all the cells to the right of and including the cell located at A. The name of the new list L2 is the function return value. The list L1 now contains all the cells to the left of and excluding the cell located at A.

NULSTL(L1,A,L2) frv: L2

This function causes the list L1 to be split into two separate lists at A, the machine address of a cell on the list L. A new list L2 is created that contains all the cells to the left of and including the cell located at A. The name of the new list L2 is the function return value. The list L1 now contains all the cells to the right of and excluding the cell located at A.

TOP(L) frv: D

This function has as its value the datum D stored in the top (leftmost) cell of the list L. The list L is not changed.

BOT(L) frv: D

This function has as its value the datum D stored in the bottom (rightmost) cell of the list L. The list L is not changed.

NTHTOP(L,N) frv: D

This function has as its value the datum D stored in the Nth cell of the list L counted from the top of the list down. The list L is not changed.

NTHBOT(L,N) frv: D

This function has as its value the datum D stored in the Nth cell of the list L counted from the bottom of the list up. The list L is not changed.

POPTOP(L) frv: D

This function returns as its value the datum D stored in the top cell of the list L and removes this cell from the list, i.e., the top cell is returned to available space and the second cell becomes the top cell.

POPBOT(L) frv: D

This function returns as its value the datum D stored in the bottom cell of the list L and removes this cell from the list, i.e., the bottom cell is returned to available space and the next-to-the-last cell becomes the bottom cell.

MADOBJ(D,L) frv: A or 0

This function searches the list L for the datum D. The function return value is the machine address A of the first occurrence of D. If D is not on L, the function return value is 0 (zero). The list L is not changed by this function.

SUBST(L or D1, A) frv: D2

This function replaces the datum D2 stored in the cell located at machine address A by either the list name L or the datum D1. The datum D2 is returned as the function value. If the first argument is L, the list L is "placed" in the cell located at A.

SUBSTP(L1 or D1, L2) frv: D2

This function replaces the datum D2 stored in the top cell of L2 by either the list name L1 or the datum D1. The datum D2 is the function return value. If the first argument is L1, L1 is "placed" in the top cell of L2.

SUBSBT(L1 or D1, L2) frv: D2

This function replaces the datum D2 stored in the bottom cell of L2 by either the list name L1 or the datum D1. The datum D2 is returned as the function value. If the first argument is L1, the list L1 is "placed" in the bottom cell of the list L2.

REMOVE(A) frv: D

This function removes the cell located at the machine address A. The function return value is the datum D that was stored in the cell.

REPLAC(L1,L2,L3) frv: L1

The lists L2 and L3 contain strings of characters. This function replaces every occurrence of the string in L2 that occurs in the list L1 by the string in L3. The value of the function is the name of the list L1.

EXISTS(Y) frv: B

This function returns the Boolean value 1 if the identifier Y has been designated as a variable of the program (usually done by an assignment statement (e.g. a=1) or an OWN statement, see page 20). Otherwise, the function returns the Boolean value 0.

ATOM(Y) frv: B

This function returns the Boolean value 0 if the identifier Y is the name of a list; otherwise, the function returns the Boolean value 1.

FSTATE(D1,D2) frv: B

This function returns the Boolean value 1 if there is a file on this disk whose first name is indicated by D1 and whose second name is indicated by D2. If no such file exists, the function return value is the Boolean value 0. The file is not changed.

Note: The following four functions provide for different types of comparisons between lists. In the first three, ALL, ANY, and NONE, the list L2 is linearized (see page A4) before comparing. This means that words within parentheses are included for the comparisons. The list L1 should not contain any parentheses.

ALL(L1,L2) frv: D or 'TRUE

This function searches the list L2 to determine if all the words on the list L1 are included. If this is the case, the function return value is the word TRUE. If all are not included, the function return value is the first word D on L1 not found on L2. The lists are not changed.

ANY(L1,L2) frv: D or 'FALSE

This function searches the list L2 to determine if any of the words on the list L1 are included. If this is the case, the function return value is the first word D on L1 found on L2. If no L1 words are found on L2, the function return value is the word FALSE. The lists are not changed.

NONE(L1,L2) frv: D or 'TRUE

This function searches the list L2 to determine if none of the words on the list L1 are included. If this is the case, the function return value is the word TRUE. If any words are found, the function return value is the first word D on L1 found on L2. The lists are not changed.

LSTDIF(L1,L2) frv: 0 or -1

This function compares the two lists L1 and L2 to see whether or not they are identical. If the lists are identical, including all sublists and sublists of sublists, etc., the function return value is 0 (zero). Otherwise, the lists are different, and the function return value is -1 (minus one). The lists L1 and L2 are not changed.

DESCRIPTION LIST FUNCTIONS

A description list is a list that is associated (by a link in the header) with another list. In most of the functions that deal with description lists, it is assumed that the description list is made up of pairs of data, the first datum of the pair being called the attribute and the second being called the value.

MAKEDL(L1,L2) frv: L2

This function makes the list L1 a description list of the list L2--i.e., it associates L1 with L2 through a link. The function return value is the name of the list L2. The contents of the lists L1 and L2 are not changed.

NODLST(L1) frv: L2 or 0

This function removes the description list of the list L1--i.e., the links between the lists are removed. The contents of the lists are not affected by this function. The function return value is the name of the description list L2. If L has no description list, the function return value is 0 (zero).

LSTNAM(L1) frv: L2 or 0

This function determines if the list L1 has a description list. If it does, the name of the description list L2 is the value of the function. If L1 does not have a description list, the value of the function is 0 (zero). Neither L1 nor L2 is changed by this function.

ITSVAL(D1,L) frv: D2 or 0

This function returns as its value the value D2 of the attribute D1 in the description list associated with the list L if that attribute is on the description list. Otherwise, the function return value is 0 (zero). Neither list is changed.

VAL(D1,L) frv: D2 or 'NONE

This function returns as its value the value D2 of the attribute D1 in the list L. If the attribute D1 is not present, the function return value is the word NONE. The list L is not changed. Note that this function does not refer to a description list (as ITSVAL does), but still assumes that L is a list of pairs of data.

NEWVAL(D1,D2,L) frv: D3 or 0

This function returns as its value the value D3 of the attribute D1 in the description list associated with the list L. The datum D2 replaces the datum D3 as the value of the attribute D1. If D1 is not on the description list, it is put there with D2, and if there is no description list, one is created. In these cases, the function return value is 0 (zero). The list L is not changed.

NOATVL(D1,L) frv: D2 or 0

This function deletes the attribute D1 and its value D2 from the description list of the list L. The function return value is D2. If D1 is not found, or if there is no description list, the function return value is 0 (zero). The list L is not changed.

SEQUENCE READER FUNCTIONS

The following functions are concerned with the sequence reader. They enable the scriptwriter to examine single cells on a list. The sequence reader is essentially a pointer that points to a specific cell. It can be moved right (down) or left (up) on the list one cell at a time. Every list has a special cell called the header, which contains the machine address of the top and bottom cells of the list. When a sequence reader is moved up past the top cell or down past the bottom cell, it points at the header. Thus, a list can be considered to be circular, with the header cell linking the top and bottom cells.

SEQRDR(L) frv: R

This function initializes a sequence reader of the list L. The function return value is the name of the sequence reader R, which initially points at the header of the list L. The list L is not changed.

SEQLR(R) frv: D or 'NIL

This function moves the sequence reader R one cell to the right (down) in the list of which it is a reader. The function return value is the datum D in the cell to which R now points. If R was initially pointing at the header, it now points to the top cell. If R was initially pointing at the bottom cell of the list, it now points to the header, and the function return value is the word NIL. The list L is not changed.

Example: One of the most important uses of the sequence reader mechanism is in locating a specific datum in a list. This can be done by the following OPL program. This program checks the list LST for the word KITTEN. If it is in the list, the system goes to the label FOUND. If it is not in the list, the system goes to the label NOTFND. Of course, all the identifiers are mnemonic.

```
      RDR = SEQRDR(LST),
*T    DATUM = SEQLR(RDR),
      IF DATUM .E. 'KITTEN THEN GOTO FOUND :
      IF DATUM .E. 'NIL THEN GOTO NOTFND ELSE GOTO T :
```

SEQLL(R) frv: D or 'NIL

This function moves the sequence reader R one cell to the left (up) in the list of which it is a reader. The function return value is the datum D in the cell to which R now points. If R was initially pointing at the header, it now points at the bottom cell of the list. If R was initially pointing at the top cell on the list, it now

points at the header, and the function return value is the word NIL. The list L is not changed.

SEQPTR(R) frv: A

This function returns as its value the machine address A of the cell to which the sequence reader R points. The reader R is not moved. The list L is not changed.

Example: In a few functions, such as REMOVE and SUBST, a machine address A is a variable of the function. The function SEQPTR is very useful in connection with these functions since its value is a machine address A. The sequence reader mechanism and its associated functions can be used to locate a certain datum in a list and the function SEQPTR used to obtain the machine address of the cell that contains the datum. Then the functions mentioned above may be used with the obtained machine address. If, in the example above, it was desired to substitute the word CAT for the word KITTEN, this could be done by the following OPL program under the label FOUND.

```
*FOUND  ADR = SEQPTR(RDR),
        SUBST('CAT,ADR),
```

This program section was written only as a demonstration of the use of the sequence reader. The commands

```
ADR = MADOBJ('KITTEN,LST),
IF ADR .E. 0
  THEN GOTO NOTFND
  ELSE SUBST('CAT,ADR) :
```

makes the same substitution in a simpler and faster way.

KEYWORD AND SENTENCE ANALYSIS FUNCTIONS

The following functions allow the programmer to work directly with keyword structures, decomposition and reassembly rules, and other operations that are part of the ELIZA and CTSS system.

A keyword structure contains the precedence number, the keycode, the decomposition rules, the reassembly rules, and the labels that are associated with a keyword.

In the process of analyzing an input, the keyword structures of the keywords in the input are "placed" in a list in the order of their occurrence in the input. This list is called a keystack. When ELIZA has generated a keystack in the normal input analysis, its name is KA(N), where N is the group number.

ADDKEY(SA(N), L) frv: L

This function adds a keyword to the script located in group area SA(N), where N is the group number. The list L must contain a keyword or a substitution in the proper format (see pages 11 and 13). The added keyword does not become a permanent part of the script, but is temporary each time the script is called and the function executed. The list L is not changed.

KEY(L1, SA(N), L2) frv: L3 or 0

This function finds the keywords in an input string. The computer scans the list L1 for the keywords of the script located in group area SA(N), where N is the group number. The keys found are stacked in the keystack L2 in the order of their occurrence. The name of the list L3, which contains the keyword structure with the highest precedence number of the keys found in the scan, is returned as the value of the function. If no key is found, 0 (zero) is the function return value.

WASKEY(D,L1) frv: L2 or 0

This function checks the keystack L1 for the keycode D (the code word of six letters or less lying between the keyword and the first decomposition rule--see page 11). If the keycode is found, the keyword structure is removed from the keystack and the name of the list L2, which contains the keyword structure of the removed key, is the value of the function. If the keycode is not found, 0 (zero) is the function return value.

HIRANK(L1,N1,N2) frv: L2 or 0

This function searches the keystack L1 according to precedence number (see page 11). There are three different types of search available depending upon the value of N2.

a) N2 = -1
The first keyword with a precedence number greater than or equal to N1 is removed, and the name of the list L2, which contains the corresponding keyword structure, is the value of the function.

b) N2 = 0
The first keyword with a precedence number greater than or equal to N1 is located. The stack is split at this point with all following keys remaining on the stack. The one found and all previous keys are removed. The name of the list L2, which contains the keyword structure of the keyword found, is the value of the function.

c) N2 = 1
The first keyword with a precedence number greater than or equal to N1 is located. The name of the list L2, which contains the keyword structure of the keyword found, is the value of the function. Nothing is removed.

In all cases, 0 (zero) is the function return value if no keyword meeting the specifications is found. Also, N1 = 0 is a special case. This is interpreted as the largest possible precedence number and causes a search of the entire keystack. Thus, the name of the list containing the keyword structure of the highest ranking keyword is the value of the function in this case.

MATCH(L1,L2,L3) frv: L3 or 0

This function applies the decomposition rule contained in the list L1 to the list L2. The resulting decomposition list is "inserted" on the bottom of the list L3. The function return value is the name of the list L3, unless there is no match, in which case 0 (zero) is returned as the value of the function. Neither L1 nor L2 is changed.

ASSMBL(L1,L2,L3) frv: L3

This function applies the reassembly rule contained in the list L1 to the decomposition list L2. The resulting reassembly list is "inserted" on the bottom of the list L3. The name of the list L3 is returned as the value of the function. Neither L1 nor L2 is changed.

INDICATOR FUNCTIONS

When words or data are stored in the cells of a list, there are a few indicators in the cell structure that describe the contents of the cell. One indicator, called the alpha-numeric indicator, indicates whether the datum in the cell is alphabetic or numeric. This indicator is 0 (zero) if the datum is numeric, 1 (one) if the datum is alphabetic, and 3 if the datum is alphabetic and part of a longer word (see next paragraph).

MRKIND(0 or 1 or 3, A)

This function sets the value of the alpha-numeric indicator of the cell located at the machine address A to either 0 or 1 or 3.

INDCTR(R) frv: 0 or 1 or 3

This function returns the value, which is either 0 or 1 or 3, of the alpha-numeric indicator of the cell to which the sequence reader R is pointing. If the value is 0 or 1, they are Boolean values. The indicator is not changed.

Another separate indicator, called the word-length indicator, indicates whether or not the datum is part of the datum in the cell following it in a list. Words, in the grammatical sense, are stored in the computer in six letter chunks. Each chunk is a datum, and is stored in a single cell (see page 31). If a cell has a negative sign in the indicator, its datum has been marked as part of the datum in the cell that follows it. If a cell has a positive sign, its datum has been marked as separate from the datum in the cell that follows it, which means that it is the end of a grammatical word. For example, when the word "weekday" is stored in the computer, it is stored as: -weekda +y . When the words "week day" are stored, they are stored as: +week +day . (The + and - signs indicate the status of the word-length indicator, and are not actually stored in the cell as data.) If a word is less than six letters long, as indicated in the list by a blank space, a comma, a period, or any other character not alphabetic or numeric, the whole word is stored and the rest of the six spaces filled in with blanks (which are not printed out).

MRKPOS(A) frv: A

This function marks the word-length indicator of the cell located at the machine address A with a positive sign. This is the same as stating that its datum is to be considered as the end of a grammatical word, and separate from the datum in the cell that follows it.

MRKNEG(A) frv: A

This function marks the word-length indicator of the cell located at the machine address A with a negative sign. This is the same as stating that its datum is to be considered as part of the datum in the cell that follows it.

Note: The sign of the word length indicator of a cell is the same as the sign of a sequence reader that points at that cell. The following program section uses this fact:

```
        SR=SEQRDR(L23),
        NUM=0,                  //WORD COUNTER//
*LA     DA=SEQLR(SR),
        IF DA .E. 'NIL          //END OF LIST//
          THEN TYPE('THERE ARE ' NUM, 'WORDS. ').:
        IF SR .G. 0             //SIGN OF SEQUENCE READER//
          THEN NUM=NUM+1 :      //END OF ANOTHER WORD//
        GOTO LA .
```

This program section counts the number of grammatical words (which includes periods and commas) in the list L23.

A third separate indicator, called the list-mark indicator, is used only when the content of the cell is the name of a list. The indicator is either 0 (zero) or 1 (one). It indicates nothing about the list, but can be used by the programmer as an indicator of whatever he chooses. That is, the list-mark indicator reflects no information about the internal structure of the list, except as the programmer sets and interprets the value of the indicator.

MRKLST(0 or 1, L)

This function sets the value of the list-mark indicator of the list L to either 0 or 1.

LSTMRK(L) frv: 0 or 1

This function returns the value of the list-mark indicator of the list L. The indicator is not changed.

MATHEMATICAL FUNCTIONS

The function return values of the following OPL functions are the identifiers to the left of the equals sign.

F2 = SIN(F1)	F2 equals the sine of F1
F2 = COS(F1)	F2 equals the cosine of F1
F2 = TAN(F1)	F2 equals the tangent of F1
F2 = COT(F1)	F2 equals the cotangent of F1
F2 = ARCSIN(F1)	F2 equals the inverse sine of F1
F2 = ARCCOS(F1)	F2 equals the inverse cosine of F1
F2 = ARCTAN(F1)	F2 equals the inverse tangent of F1
F2 = TANH(F1)	F2 equals the hyperbolic tangent of F1
F2 = SQRT(F1)	F2 equals the square root of F1
F2 = SQUARE(F1)	F2 equals the square of F1
F2 = CUBE(F1)	F2 equals the cube of F1
F2 = LOG(F1)	F2 equals the natural logarithm of F1
F2 = EXP(F1)	F2 equals the number e raised to the power of F1
F2 = ABS(F1)	F2 equals the absolute value of F1
F2 = MODULO(F1,N)	F2 equals the modulus of F1 to the base N
F = MAX(F1,F2)	F equals the larger of the pair F1 F2
F = MIN(F1,F2)	F equals the smaller of the pair F1 F2
N = INTGER(F)	N equals the integer part of F, returned as an integer
F2 = INTPRT(F1)	F2 equals the integer part of F1, returned as a floating point number
F = RANDOM(0)	This function returns a random fraction F in the interval (0,1)
F = RANSET(F)	This function sets F as the initial point of the random number generator,

and is used in order that different sequences of random numbers can be generated with the function RANDOM

The following operators may also be used.

+	add
-	subtract
*	multiply
/	divide
**	raise to the power of

BOOLEAN FUNCTIONS

Certain functions make use of Boolean variables and Boolean algebra. A Boolean variable has one of two values: 1 (true) or 0 (false). The following Boolean operators may be used in Boolean expressions.

.E.	equal to
.LE.	less than or equal to
.GE.	greater than or equal to
.NE.	not equal to
.L.	less than
.G.	greater than
.AND.	and
.OR.	or

A space should precede and follow the completed expression in order to avoid confusing the periods with numbers preceding or following the expression. For example, write (X .E. 1), not (X.E.1).

The following three OPL functions perform Boolean operations.

B3 = AND(B1,B2)

This function returns the value B3 of the logical operation and on the Boolean values B1 and B2. B3 = 1 if and only if B1 = 1 and B2 = 1; otherwise, B3 = 0.

B3 = OR(B1,B2)

This function returns the value B3 of the logical operation or on the Boolean values B1 and B2. B3 = 0 if and only if B1 = 0 and B2 = 0; otherwise, B3 = 1.

B2 = NOT(B1)

This function returns the value B2 of the logical operation not on the Boolean value B1. If B1 = 0, B2 = 1. If B1 = 1, B2 = 0.

IF

The IF statement has the following format-

IF Boolean THEN program ELSE program :

The expression immediately following the IF must be either true or false (Boolean 1 or 0). If it is true, the program immediately following the THEN is executed; if it is false, the program immediately following the ELSE is executed. The IF statement must be terminated by a colon (:). After the appropriate program is executed, the system goes to the line following the IF statement, unless the program executed caused the system to go to another part of the script. The following IF statement is an example of an algebraic sign determination.

```
IF X .L. 0 THEN X= -1*X ELSE GOTO A :
```

If X is less than zero, the sign of X is changed, and the system goes to the next statement after the colon. If X is not less than zero, the system goes to the label A. If it is desired only to transfer control to the next line when the IF statement is false, the "ELSE program" may be omitted, as:

```
IF X .L. 0 THEN X= -1*X :
```

IF statements may be used within the program of an IF statement, providing they are properly terminated by colons.

FOR

The FOR statement has the following format-

FOR program STEP program UNTIL Boolean DO program :

The FOR statement is basically designed to exectue a certain program a certain number of times, depending upon a test and upon a certain variable that is incremented after each execution. The program following FOR usually initializes a variable, such as I = 1, and is executed only once. The program following STEP usually contains an integer that is the amount that the variable is to be incremented each time. The expression following UNTIL must be a Boolean, which is tested each time the variable is increased. If the Boolean is true, the system goes to the statement after the colon. If the Boolean is false, the program following DO is executed. The order of execution of the parts of the FOR statement is as follows:

```
FOR program
UNTIL Boolean (true or false?)
DO program
STEP program
UNTIL Boolean (true or false?)
DO program
STEP program
UNTIL Boolean (true or false?)
DO program
STEP program
etc., until the Boolean is true.
```

The following FOR statement is an example that computes the factorial of a positive integer N, and returns the result as the value of F.

```
FOR I=1, F=1 STEP 1 UNTIL I .G. N DO F=I*F :
```

If N was equal to 3, the statement would work in the following manner. First, I and F are set equal to 1. I is identified as the variable to be incremented since its assignment statement comes first. Next, I .G. N is checked and since 1 .G. 3 is false, F is set equal to 1*1 which is 1. Next, I is increased by 1, which makes its value 2. The whole process is now repeated. Since 2 .G. 3 is false, F is set equal to 2*1 which is 2. I is increased by 1 to 3. Again, since 3 .G. 3 is false, F is set equal to 3*2 which is 6. I is increased by 1 to 4. This time 4 .G. 3 is true, and the system goes to the next statement. The value of F is 6, which is 3 factorial.

If it is desired to have a program following STEP and still increment the variable, this may be done by putting the increment as a single number as the first statement of the program. For example, the factorial statement could have been written as:

```
FOR I=1, F=1 STEP 1, F=I*F UNTIL I .E. N DO :
```

FOR statements may be used within the program of a part of a FOR statement, providing they are properly terminated by colons.

READING, WRITING, AND PRINTING FUNCTIONS

The following functions deal with reading, writing, and printing. For a more detailed description of some aspects of these functions, see Chapter 7, "Reading, Writing, and Printing", which begins on page 54.

TYPE(Y1,Y2,...,Yn)

The TYPE function can be used for many purposes. It can be used for both printing and writing, since the output is controlled by the code words on the list STOUT (see pages 55 and 57). It can also be used to modify the format of the output by including spaces, tabs, and carriage returns. The number of arguments Y1,Y2,...,Yn is indefinite but not infinite. Each Y can be one of the following identifiers, which will cause the indicated output.

Argument	Output
L	the contents of the list L
'SPACE(N) '	N spaces
'TAB(N) '	N tabs
'LINE(N) '	N carriage returns
'comment '	the characters between the apostrophes ('), in this case: comment
D	D the value of the datum D
'comment ' D	comment the value of the datum D
(function)	the value of the function (if a list)
	function the value of the function (if a datum)

The space before the last apostrophe is necessary in the arguments that have apostrophes.

The following section of an OPL program with its output demonstrate the use of the various arguments of the TYPE function.

The program section:

```
STOUT='(CONCAT),
L1='(THE FUNCTION TYPE PRINTS LISTS,),
L2='(ALSO SPACES),
L3='(TABS),
L4='(AND CARRIAGE RETURNS),
L5='((FUNCTIONS.)),
D=1,
TYPE(L1, 'AND COMMENTS '),
TYPE(L2, 'SPACE(4) ', L3, 'TAB(2) ', L4, 'LINE(2) '),
TYPE('AND DATA ' D, 'LINE(1) ', D),
TYPE('AND THE VALUE OF ', (TOP(L5))),
```

The output:

```
THE FUNCTION TYPE PRINTS LISTS, AND COMMENTS
ALSO SPACES     TABS                    AND CARRIAGE RETURNS
AND DATA  1.0
D         1.0
AND THE VALUE OF FUNCTIONS.
```

TXTPRT(L1, L2 or 0) frv: L1

This function causes the computer to print the contents of the list L1 as a linear text string according to the code words on the list L2 (see page 55). If the second argument is 0 (zero), the list L1 is simply printed on the teletype.

PRTLC(D1,D2,D3)

This function prints the disk file whose name is specified by the data D3 D2 from the archive file whose name is specified by the data D1 D2 (see section AH.4.01 of the CTSS Programmer's Guide). The files must be created and edited using TYPSET instead of EDL (see section AH.9.01 of the CTSS Programmer's Guide). This function is useful for printing large amounts of material, with both upper and lower case characters (see page 34).

PRTUC(D1,D2,D3)

This function prints the disk file whose name is specified by the data D3 D2 from the archive file whose name is specified by the data D1 D2 (see section AH.4.01 of the CTSS Programmer's Guide). The files must be created and edited by using EDL, which means that all letters will be capitals, and that certain characters cannot be used.

DSKLST(D1,D2,L) frv: L or 'DONE or 'GOOF

This function reads lists from the disk file whose name is specified by the data D1 D2. The file must consist of one or more lists (indicated by parentheses) and must be in EDL format. Each main list must start on a new line. A script is a good example of such a file, where each keyword and the program section are main lists. The first use of DSKLST opens the file and reads the first list into the list L, which is also the function return value. The next use of DSKLST returns the second list in the file, and so on. When DSKLST is used after the last list has been read, the word DSKLST is returned as the value of the function and the file DONE is closed. The disk file is not changed by the reading of its lists. If the file cannot be found, an error message is printed, and the function return value is the word GOOF.

DSKCLS(0,0,0) frv: 'DONE or 'GOOF

This function closes a file that is open for either reading or writing. It has the same effect as the code word CLOSE (see page 56). The function return value is the word DONE. If there is no open file, an error message is printed, and the function return value is the word GOOF. The arguments of the function are zeroes and must appear.

Example: The following program section checks the disk file named ALPHA BETA for a list whose top element is the word CHILD. When it is found, the list is put in the list C, and the file is closed with DSKCLS. If it is not found, the system goes to the label F.

```
*E      D=DSKLST('ALPHA,'BETA,LIST(C)),
        IF D .E. 'DONE THEN GOTO F :
        IF TOP(C) .NE. 'CHILD THEN GOTO E :
        DSKCLS(0,0,0),
```

ARCHRD(D1,D2,L) frv: L or 'DONE or 'GOOF

This function reads from an archive file on the disk (see section AH.4.01 of the CTSS Programmer's Guide). The first name of the archive file is specified by D1 and the second name must be ARCHIV. It looks for the component file whose first name is specified by D2. If it finds the file, it reads it in the same manner as DSKLST. That is, it opens the file and reads the first list into the list L on its first use, it reads the second list on its second use, and so on. The function return value is the name of the list L. When ARCHRD is used after the last list has been read, the word DONE is returned as the value of the function and the file is closed. The disk file is not changed by the reading of its lists. If either the archive file or the component file cannot be found, an error message is printed, and the function return value is the word GOOF.

SCRIPT(N, D1 or L) frv: D1 or L or 'GOOF

This function reads the script whose name is specified by D1 from the disk into group area SA(N), where N is the group number. If the second argument is a list of the form (D2 D3), the disk file named D2 D3 is read in. If the file is not found, an error message is printed and the function return value is the word GOOF. See page 19 for a full description of changing scripts.

DELSCR(N)

This function deletes the script stored in group area SA(N), where N is the group number. DO NOT delete a script in the group area that is presently active.

LOAD(D,L1,L2) frv: L2 or 'GOOF

This function reads a number of lists from the disk file whose first name is indicated by the datum D and whose second name is LIBE (for library). The list L1 should contain the first word of all the lists that should be read in. The lists are "placed" on the list L2, the function return value. This function can be used, for example, to read a number of defined functions from the disk file DEFINE LIBE (see page A29). To be made defined functions of the script, the list L2 must be evaluated. After the functions are evaluated, the contents of the list L2 are not needed, so L2 should be recreated. For example, to load RANGE and KKK, write:

```
EVAL(LOAD('DEFINE,'(RANGE KKK),LIST(TEM))),
LIST(TEM),
```

If any of the words on the list L1 cannot be found as the first word of the lists in the LIBE, a message to that effect is printed, and the function return value is the word GOOF.

READ(Y) frv: Y

This function causes the computer to print "Y = " on the typewriter without a carriage return, and waits for an input until a double carriage return is typed. The identifier Y is given the value of the input. This function can be used to read numbers or an OPL program.

RDLONL(L) frv: L

This function causes the computer to read input from the teletype into the list L. The computer will wait until a double carriage return is typed before continuing with the program.

OTHER FUNCTIONS

GOTO Y

This function causes the system to go to the label Y, which must be marked by *Y in the script. If Y is a list, indicated by parentheses, the list is executed as an OPL program section and the system will go to the label that is the result. For example: GOTO(POPTOP(DAHIN)).

QUIT(0)

This function causes the computer to leave the ELIZA system and return to the CTSS command level.

XECOM(L) frv: L

This function causes the computer to execute the CTSS commands contained in the list L. The name of the list L is the function return value. The list L is not changed.

EVAL(Y1) frv: Y2

This function evaluates the identifier Y1 in the same manner that the program is evaluated (or exectued). The function return value is the last result Y2.

TIME(0) frv: F

This function returns the time of day in seconds as a floating point number F. The time is computed to tenths of seconds, and is based on a 24 hour clock. For example, at 6 minutes and 18.2 seconds past 5 o'clock in the afternoon, F would be 61578.2 .

TODAY(L) frv: L

This function "inserts" the date and the time of day on the bottom of the list L. The date contains the month and the day, and the time is computed to tenths of minutes on a 24 hour clock. For example, at 45 minutes and 24 seconds past 8 o'clock on the evening of November 4, the function would cause the list L to be (11/4, 2045.4).

DEFINE

In using the OPL functions in a script, it is sometimes necessary to use the same set of OPL functions for a particular purpose a number of times in the program. Rather than having to write out this set of functions each time it is desired to use them, the ELIZA system has the feature of allowing the programmer to define or construct a function that will perform the set of functions when called. This is similar to a subroutine in some other computer languages.

This defined function is similar in structure to the OPL functions previously discussed (see the section of this manual on the "General Characteristics of Functions", page A1). A defined function has a specified set of variables, performs prescribed operations on them, and has a function return value. However, a defined function is different from the OPL functions in the following respects: the operations it performs are determined by a set of OPL functions; the function is defined only after the DEFINE statement is executed, but then is defined until the user quits the ELIZA system; and, most importantly, the function is defined by the writer of the script.

The format for defining a function is as follows (notation: V -- variable ; FRV -- function return value):

```
DEFINE(NAME(V1,V2,...,Vn)=
  OPL function,
  OPL function,
        .                    (this is the OPL
        .                     program that the
        .                     function performs)
  OPL function,
  FRV )
```

NAME is the name which the programmer gives the function, which must consist of six letters or less. There can be any number of variables, but there cannot be an indefinite (or infinite) number--all the variables must be specified in the DEFINE statement. All labels are local to the set of OPL functions in the DEFINE statement. If variables are to be known only to the defined function, this should be specified by OWNLIST (see page 20) or LET (see next function). If the FRV is omitted, the value of the last OPL function executed is the function return value.

The DEFINE statement is usually included in the section under the label START in the script. When the function is used in the script, it is used in the very same manner that an OPL function is used, as the following usage format indicates:

NAME(V1,V2,...Vn)

Whenever this is specified in the script, the sequence or set of OPL functions under the DEFINE statement will be performed.

A simple defined function can illustrate this process. In conversation with a student, it may be desirable to rotate among a certain number of essentially identical responses whenever the student makes a certain statement. The following function will do this:

```
DEFINE(ANSDIF(LST)=
  OWN(X),
  X = POPTOP(LST),
  TYPE(X),
  NEWBOT(X,LST),
  LST),
```

The function ANSDIF takes the top element off the list specified, prints this element, puts it on the bottom of the list, and returns the name of the list as the value of the function. Thus, the next time ANSDIF is used on this list, a new top element (originally the second element) will be printed, and so on. When ANSDIF is used, it may be used on any list that has been set up in the script. In the above function, LST is only a mnemonic identifier, and when the function is used on a list, the name of that list is used in the place of LST.

A number of useful defined functions are stored on the disk in the file DEFINE LIBE. For a description of their use see page A29.

LET

The LET statement operates in exactly the same manner as the DEFINE statement, except that the function defined is defined only for the particular script that contains the LET statement. In addition, a LET statement can be used to give values to certain variables, which will be known only to the particular script. For example:

LET (X=1) (Y=2),

If a LET statement is used within a DEFINE or a LET statement, the variables or the function that are LET will be known only within that statement.

SOME DEFINED FUNCTIONS--THE DEFINE LIBE

Since a number of defined functions are useful to many scriptwriters, there is a library of these functions stored on the disk in a file named DEFINE LIBE (on M5347 cmf101 at Comp Center). Defined functions are described on page A27, and the loading of defined functions from the LIBE is described on page A25.

The format of the functions in DEFINE LIBE is:

```
( NAME(V1,V2,...,Vn)=

    program of the function

    DLIST(description of what the function does) )
```

Each function is a list, indicated by the outer parentheses, with a description list that contains information about the function.

If the scriptwriter has a defined function that he thinks might be useful to others, he should check the DEFINE LIBE to make sure that there is no function that already does what his does, and that there is no function that has the same name as his function. Then his function, in the proper format, may be added by EDLing the DEFINE LIBE.

The DEFINE LIBE included the following functions at the time this manual was published.

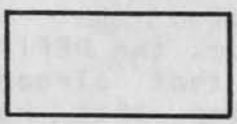
```
(RANGE(L,U)= INTPRT(L+(U+1-L)*RANDOM'(0))
        DLIST(THIS FUNCTION RETURNS A FLOATING POINT
          INTEGER RANDOMLY DISTRIBUTED OVER THE RANGE
          FROM L TO U) )

(KKK(ABC)=
        HIRANK(ABC,0,-1),
        LET(Z=0),
        Z=HIRANK(ABC,0,-1),
        NEWTOP(ELBAT,TABLE),
        IF Z .NE. 0
          THEN KEY=Z, EXP=INPUT.
          ELSE KEY='(X (0 X () NOKEY)), EXP='(). :
        DLIST(THIS FUNCTION SEARCHES THE KEYSTACK ABC
          FOR THE NEXT HIGHEST RANKING KEYWORD) )
```

FLOW CHART OF THE SYSTEM

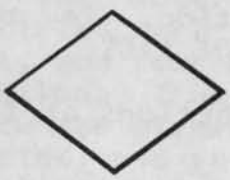
The following flow chart shows the inner workings of the ELIZA system as of January, 1968. An understanding of the details of operation should help in detecting and avoiding mistakes in programming, but it is not necessary for writing scripts.

Key to Symbols

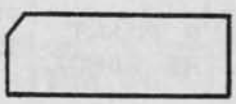
direction of operation



operations performed by the computer

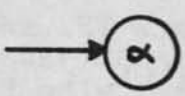


decision point for the computer

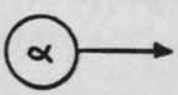


input from the console by the operator

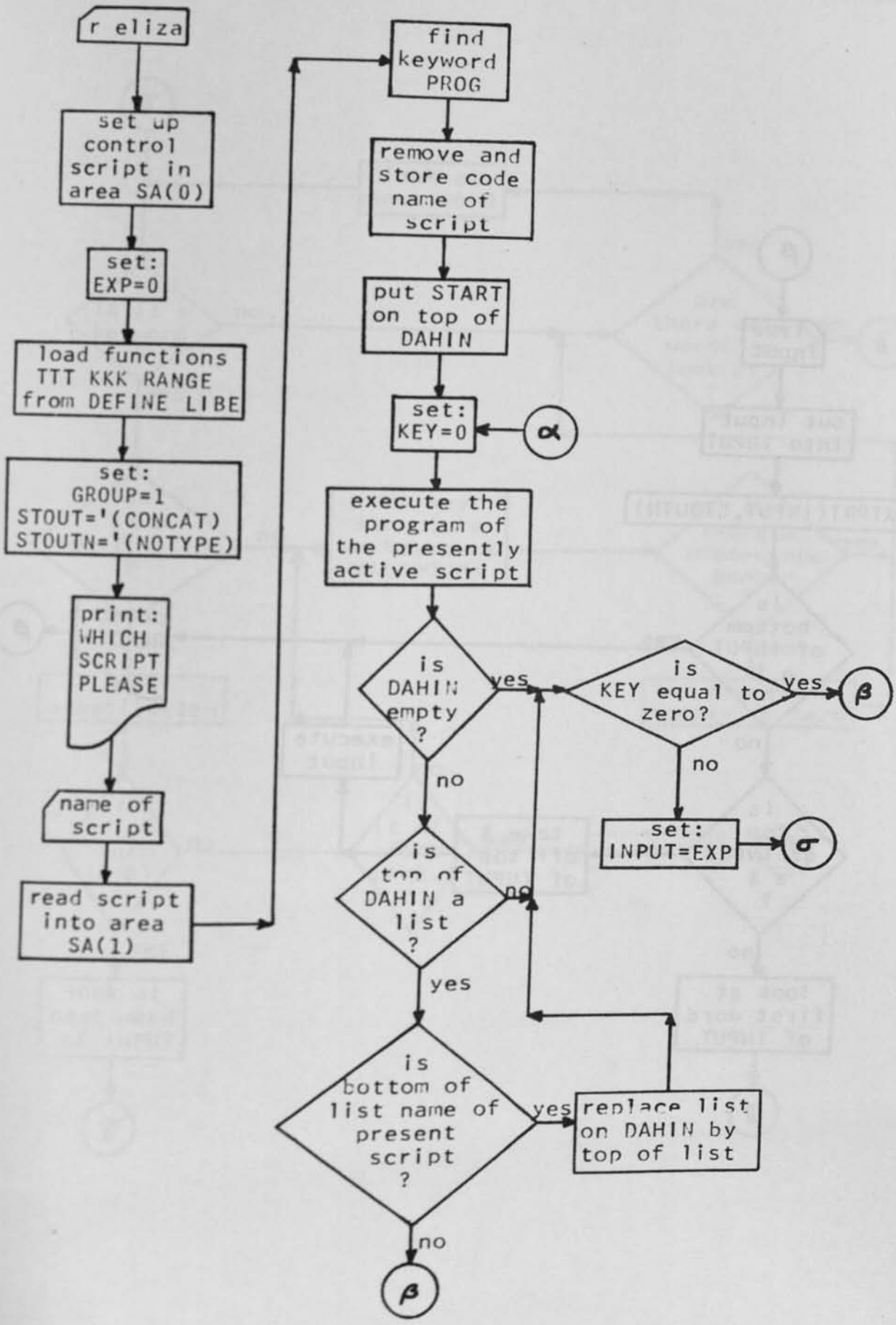
printout by the computer

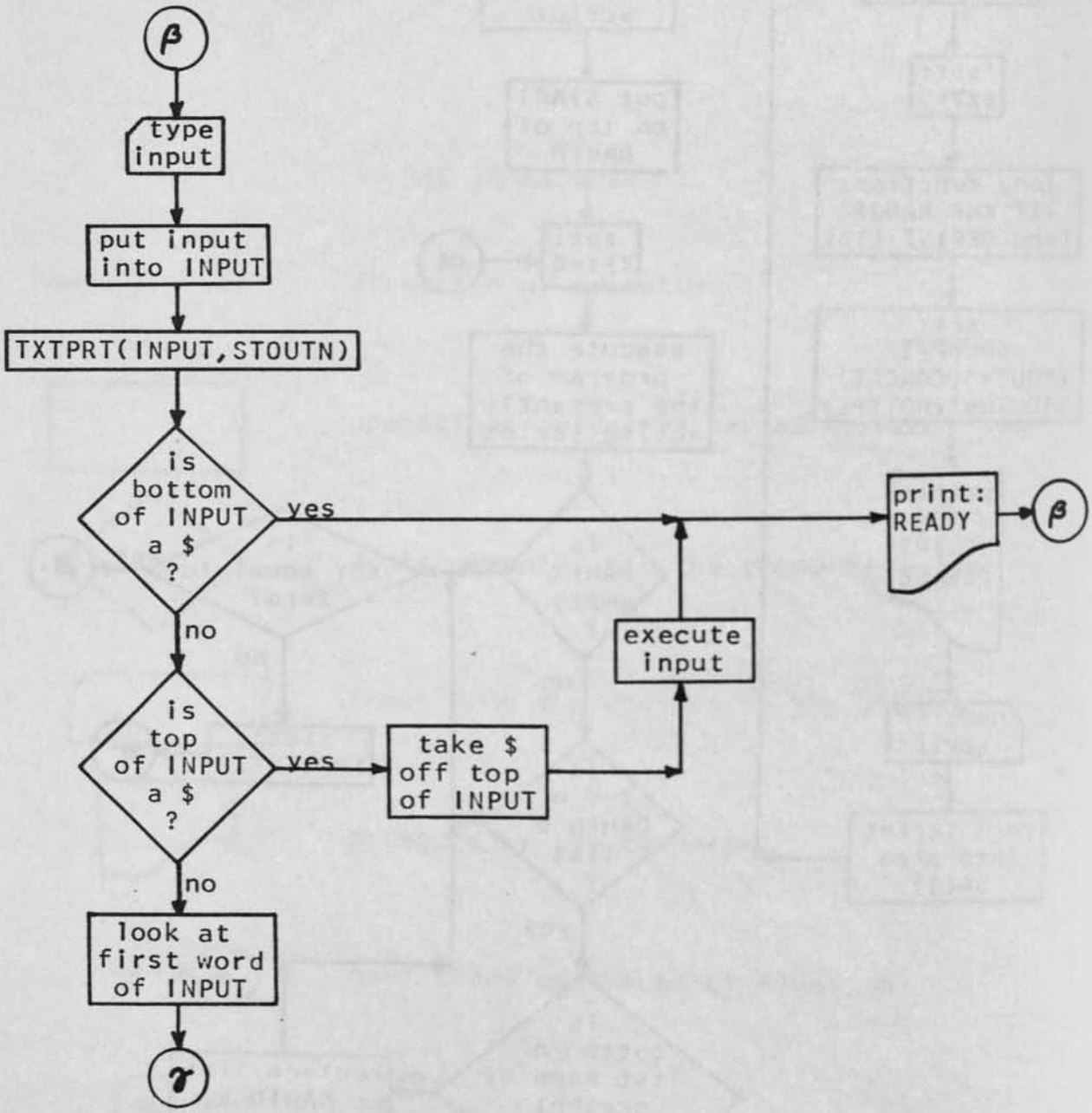


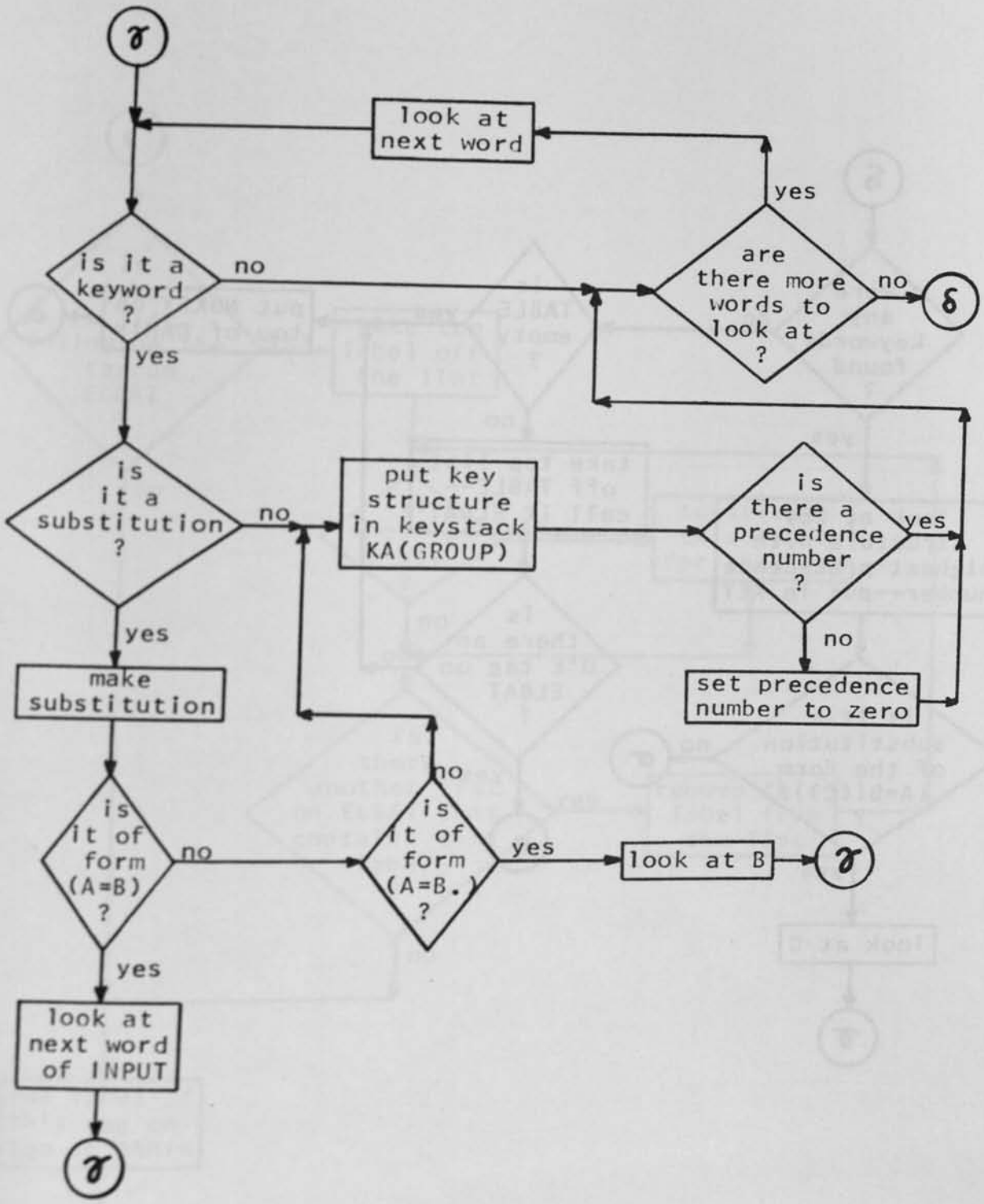
operations continued at label α

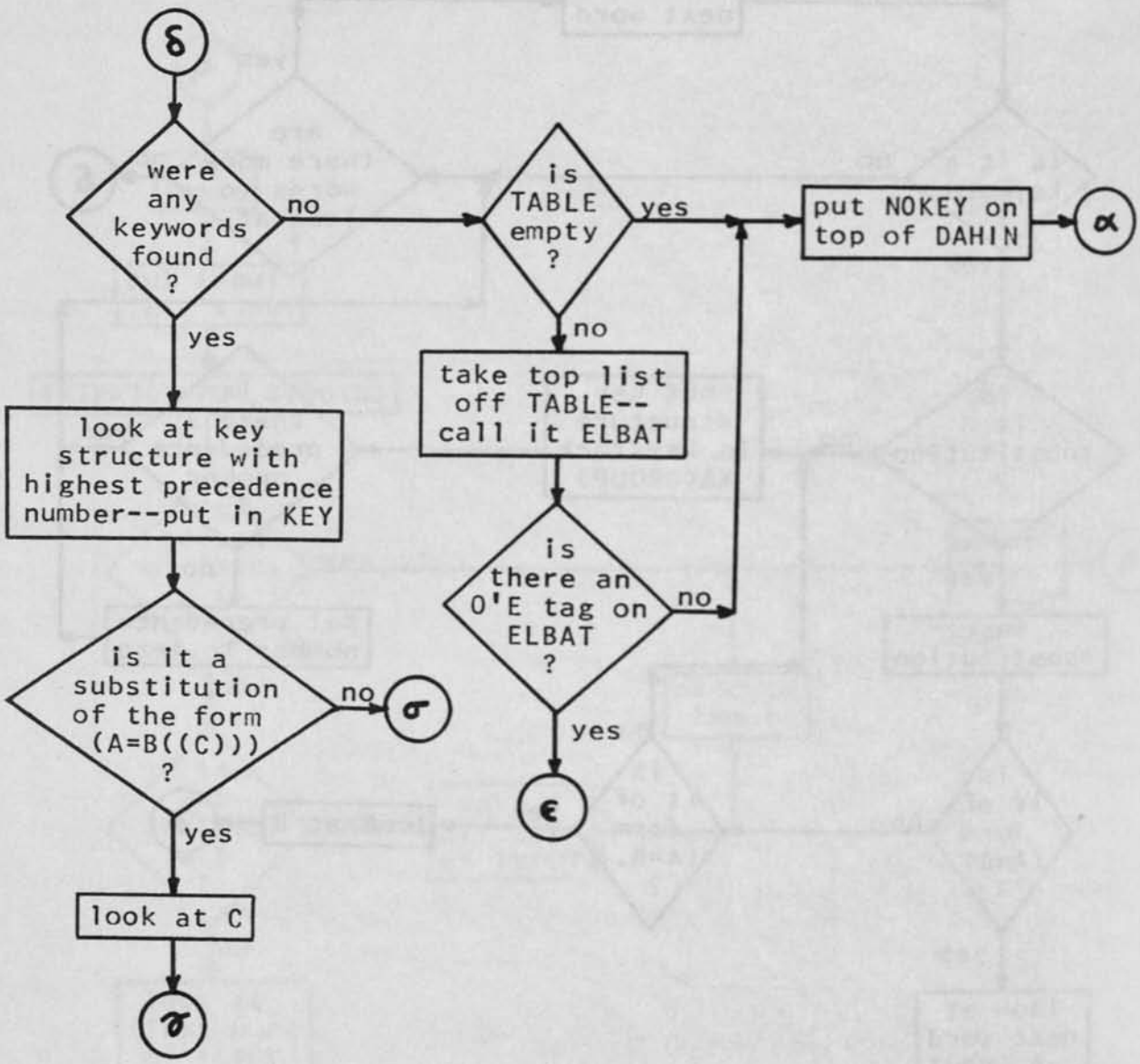


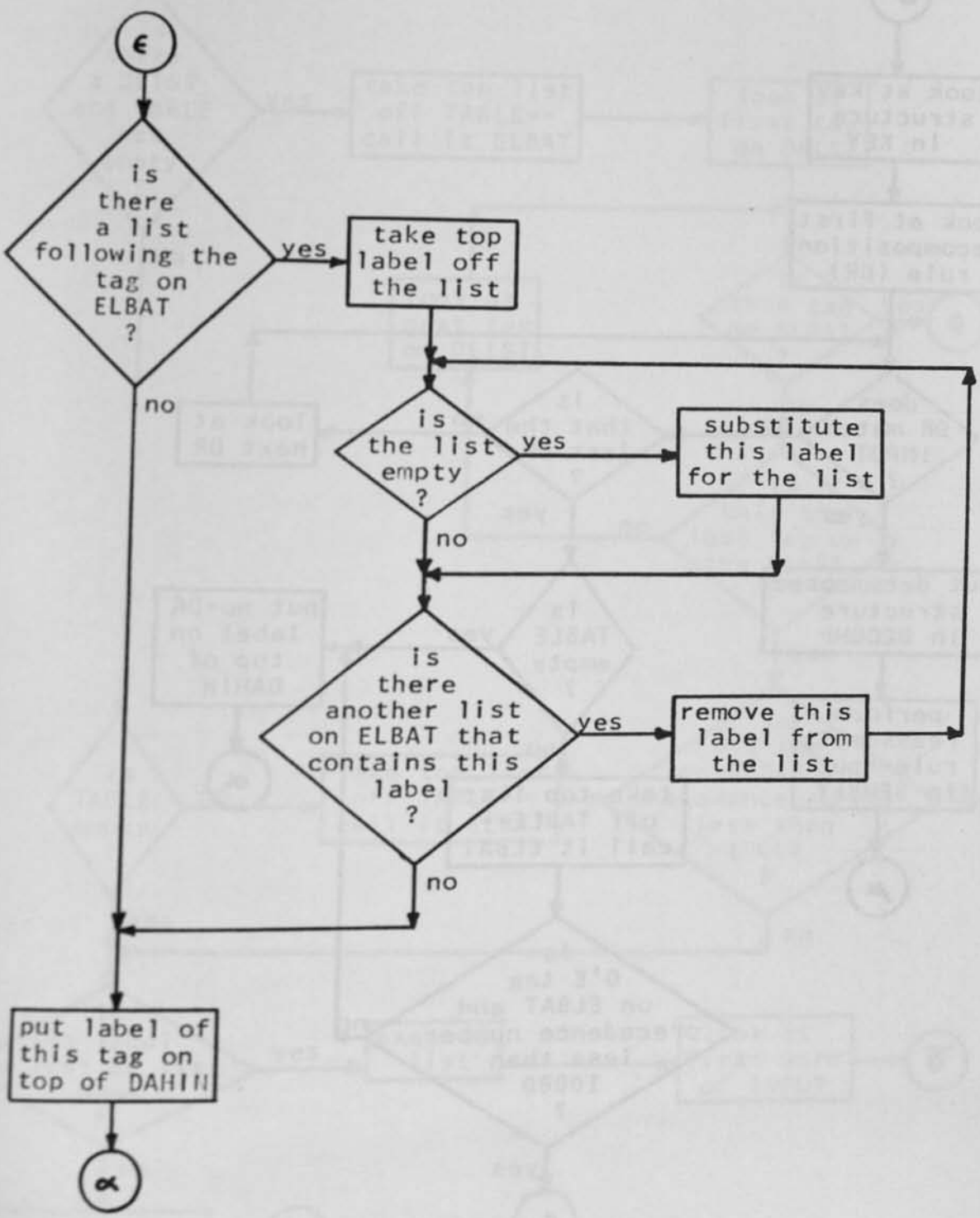
label α

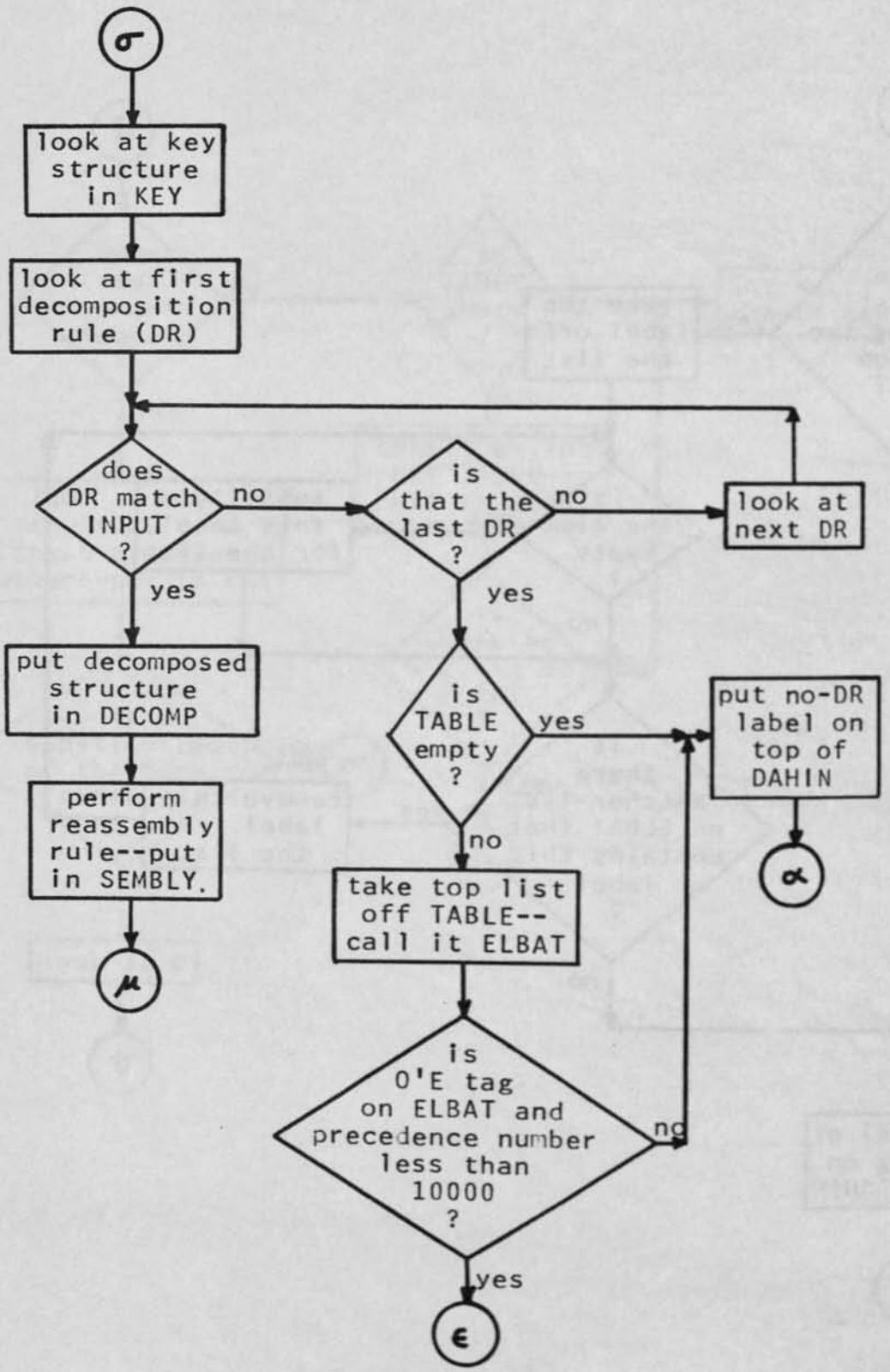

r eliza
set up control script in area SA(0)
set: EXP=0
load functions TTT KKK RANGE from DEFINE LIBE
set: GROUP=1 STOUT='(CONCAT) STOUTN='(NOTYPE)
print: WHICH SCRIPT PLEASE
name of script
read script into area SA(1)
find keyword PROG
remove and store code name of script
put START on top of DAHIN
set: KEY=0
α
execute the program of the presently active script
is DAHIN empty ?
yes
is KEY equal to zero?
yes
β
no
set: INPUT=EXP
σ
is top of DAHIN a list ?
no
yes
is bottom of list name of present script ?
yes
replace list on DAHIN by top of list
no
β

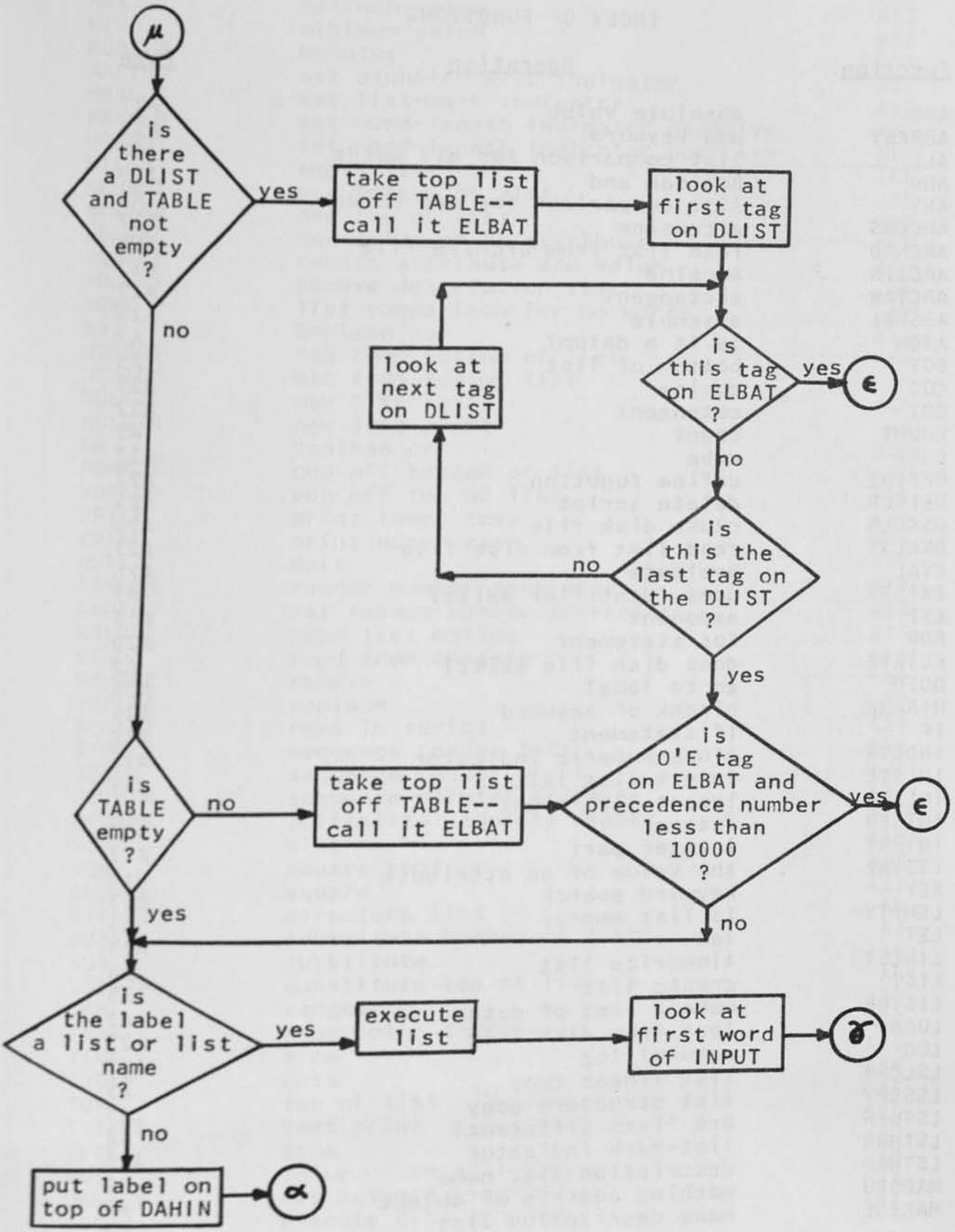

β
type input
put input into INPUT
TXTPRT(INPUT,STOUTN)
is bottom of INPUT a $?
yes
print: READY
β
no
execute input
is top of INPUT a $?
yes
take $ off top of INPUT
no
look at first word of INPUT
γ


γ
look at next word
yes
is it a keyword ?
no
are there more words to look at ?
no
δ
yes
is it a substitution ?
no
put key structure in keystack KA(GROUP)
is there a precedence number ?
yes
yes
no
make substitution
set precedence number to zero
no
is it of form (A=B) ?
no
is it of form (A=B.) ?
yes
look at B
γ
yes
look at next word of INPUT
γ


δ
were any keywords found ?
no
yes
is TABLE empty ?
yes
put NOKEY on top of DAHIN
α
no
take top list off TABLE-- call it ELBAT
look at key structure with highest precedence number--put in KEY
is there an O'E tag on ELBAT ?
no
yes
ε
is it a substitution of the form (A=B((C))) ?
no
σ
yes
look at C
τ


ε
is there a list following the tag on ELBAT ?
yes
take top label off the list
no
is the list empty ?
yes
substitute this label for the list
no
is there another list on ELBAT that contains this label ?
yes
remove this label from the list
no
put label of this tag on top of DAHIII
α


σ
look at key structure in KEY
look at first decomposition rule (DR)
does DR match INPUT ?
no
is that the last DR ?
no
look at next DR
yes
yes
put decomposed structure in DECOMP
perform reassembly rule--put in SEMBLY.
μ
is TABLE empty ?
yes
put no-DR label on top of DAHIN
α
no
take top list off TABLE-- call it ELBAT
is 0'E tag on ELBAT and precedence number less than 10000 ?
no
yes
ε


μ
is there a DLIST and TABLE not empty ?
yes
take top list off TABLE-- call it ELBAT
look at first tag on DLIST
no
look at next tag on DLIST
is this tag on ELBAT ?
yes
ε
no
is this the last tag on the DLIST ?
no
yes
is TABLE empty ?
no
take top list off TABLE-- call it ELBAT
is O'E tag on ELBAT and precedence number less than 10000 ?
yes
ε
yes
no
is the label a list or list name ?
yes
execute list
look at first word of INPUT
γ
no
put label on top of DAHIN
α

INDEX OF FUNCTIONS

MATCH	match	A14
MAX	maximum value	A17
MIN	minimum value	A17
MODULO	modulus	A17
MRKIND	set alpha-numeric indicator	A15
MRKLST	set list-mark indicator	A16
MRKNEG	set word-length indicator negative	A16
MRKPOS	set word-length indicator positive	A15
MTLIST	empty list	A3
NEWBOT	new bottom of list	A3
NEWTOP	new top of list	A3
NEWVAL	new value of an attribute	A10
NOATVL	remove attribute and value	A10
NODLST	remove description list	A9
NONE	list comparison for no words	A8
NOT	Boolean not	A19
NTHBOT	Nth from bottom of list	A6
NTHTOP	Nth from top of list	A6
NULSTL	new list left	A5
NULSTR	new list right	A5
OR	Boolean or	A19
POPBOT	pop off bottom of list	A6
POPTOP	pop off top of list	A6
PRTLC	print lower case	A23
PRTUC	print upper case	A23
QUIT	quit	A26
RANDOM	random number generator	A17
RANSET	set random number generator	A17
RDLONL	read list online	A25
READ	read from console	A25
REMOVE	remove	A7
REPLAC	replace	A7
SCRIPT	read in script	A24
SEQLL	sequence reader left	A11
SEQLR	sequence reader right	A11
SEQPTR	sequence pointer address	A12
SEQRDR	initialize sequence reader	A11
SIN	sine	A17
SQRT	square root	A17
SQUARE	square	A17
STRLST	structure list	A4
SUBSBT	substitute bottom of list	A7
SUBST	substitute	A6
SUBSTP	substitute top of list	A6
TAN	tangent	A17
TANH	hyperbolic tangent	A17
TIME	time	A26
TODAY	date	A26
TOP	top of list	A5
TXTPRT	text print	A23
TYPE	type	A22
VAL	value of an attribute	A9
WASKEY	was keyword present?	A13
XECOM	execute CTSS command	A26