APPLICATION OF A SYNTAX DRIVER TO LOGIC EQUATION PROCESSING AND DATA-CONTROL CARD SCANNING

J. A. RADER
Design Automation Group
Hughes Aircraft Company
Culver City, California

INTRODUCTION

This paper will describe the use of a syntax driver in writing a logic equation compiler (LOGCOM). By such a compiler or processor we mean a program which takes logic equations punched on cards and converts them into a tabular form which subsequently is used by other design automation programs. In the body of the paper it is assumed that the reader possesses a casual familiarity with syntax. This assumption is made mostly for organizational convenience and an appendix discussing syntax is included at the end of the paper. My terminology is summarized at the end of this appendix and the reader familiar with syntax can check there to see if his terminology and mine agree. Sample syntax applied to a specific problem is shown in figure 3 and is discussed in Section 2.

SECTION 1

DESCRIPTION OF LOGIC COMPILER PROGRAM (LOGCOM)

The purpose of our logic equation processor (hereafter referred to as LOGCOM) is really two-fold. First of all LOGCOM compiles logic equations into a set of tables which are then used by the rest of the programs in our Design Automation system. Secondly, it prints a document called a logic release which lists the equations and allows considerable flexibility for documentation purposes.

A brief description of this flexibility follows. The logic designer, by placing the appropriate character in column one, can effect a single spaced or double spaced line or eject to a new page. He can also specify an entire line as a comment line with the same carriage control as for an equation line. Comments can be placed on the same line as an equation by preceeding them with a dash. Provision is also made for a title and subtitle on each page and these can be changed at any point in the listing. Page numbering is automatic and at the logic designer's option can be started with an integer other than one. An equation can be extended over numerous lines or two equations may be written on one line. Just to the right of each card image appears an alter number which is generated by the program. LOGCOM writes a tape containing images of the cards each time it is run and this tape can be saved and used as input in place of cards the next time the program is run. The alter numbers mentioned above are provided for update or alter

purposes. Also available with the logic release are lists of signals generated but not used, undefined signals plus a complete usage list for each signal and an index of equation name versus alter number.

One last feature that need be mentioned is the ability to handle common collector sequences. A string of signals separated by commas and enclosed in parentheses is recognized as a common collector sequence and is replaced by the generated signal, X.nnnn, where nnnn is an integer unique to each distinct common collector sequence. An example is included in figure one. The line with alter number zero is generated by LOGCOM for the logic designer and shows the common collector sequence replaced by its associated signal X.nnnn.

The reason for the use of syntax was the flexibility it allows in the achievement of task one. However, task two is described because the existence of task two complicates the achievement of task one as equations written in relatively free format must be handled correctly and large amounts of extraneous data must be ignored. Essentially, an awareness of task two is an awareness of the environment in which task one must be performed.

The explicit role of the syntax in building the LOGCOM tables is to construct simple equations from the equations as they are written by the logic designer. A simple equation is a succession of six character words consisting of a counter, an equation name and the terms which appear on the right side of the equation with that name. (See Figure 2). The counter specifies how many right side terms appear in the string. A FORTRAN program then actually builds the LOGCOM tables from the complete set of simple equations which are input to it. If any common collectors were used by the logic designers, simple equations for the generated common collector equations will be included in the complete set of simple equations. The portion of the program which scans cards and produces simple equations is called the scanner-processor, and it along with the CONTROLS-processor are the portions of LOGCOM which are syntax directed. The CONTROLS-processor reads a variety of control cards, performs updates to the input tape (when selected) and checks to see what options have been selected by the logic designer. This CONTROLS section will be discussed later.

WASHINGTON,D.C. - JULY 1968

```
DATA AND CLOCK MATRIX                                                    98

  BTRAN1 = SIDORD          - NOT RESET AND HOLD DO MATRIX (EXTERNAL)     99

  ND001T = T.0001 * BD0F/C    - DATA BIT 01 ENABLED AT DA1               100

  BD001T = ND212R * BD134K * ND001A * ND001C                             101

         * ND001F * (NC302A,NC303A,NC304A,NC302T)                        102

         * ND001F *          X.0011                                      0

  B.0002 = ND001F * SIDORR                                               103

CONTROL LOGIC                                                            104

  JW01/C = CW01..          JW01/P = RW01/3                               105
  JW01/R = NW01/A * NL023P                                               106

  NL001.D= BD06LT * JW01/T * NL001A * NC001A      CLEAR SELECT           107

                                                                         108

                          FIGURE 1                                       109
    PARTIAL PAGE OF LOGIC EQUATIONS AS WRITTEN BY A LOGIC DESIGNER       110
    AND AS THEY APPEAR ON A LOGIC RELEASE                                111

                                                                         112

1BTRAN1SIDORD                                                            113
2ND001TT.0001BD0F/C                                                      114
6BD001TND212RBD134KND001ANDD01CND001FX.0011                              115
2B.0002ND001FSIDORR                                                      116
1JW01/CCW01..                                                            117
1JW01/PRW01/3                                                            118
2JW01/RNW01/ANL023P                                                      119
4NL001.BD06LTJW01/TNL001ANC001A                                          120
    .                                                                    121
    .                                                                    122
    .                                                                    123
4X.0011NC302ANC303ANC304ANC302T                                          124

                                                                         125

                        - FIGURE 2 -                                     126
    SIMPLE EQUATIONS BUILT FROM LOGIC EQUATIONS IN FIGURE 1              127
```

SECTION 2

SIMPLE EXAMPLE OF A SYNTAX-DIRECTED LOGIC EQUATION SCANNER-PROCESSOR

The first four lines of syntax in Figure 3 will scan and process logic equations satisfying the following restrictions.

1) An equation consists of a three character equation name, followed by an equals sign which is then followed by a string of three character signal names which are separated by asterisks.

2) Each equation is terminated by the character @ which is reserved for this purpose.

3) Blanks may not be imbedded in the signal names or equation name but may occur anywhere else.

4) Comments are not permitted.
For sample equations obeying these rules see Figure 3.

The semantic routines used by these first four lines perform the following tasks:

(INITIL) - performs initialization.
(WRAPUP) - terminates the scanner-processor and calls the table building subroutine.
(SIGNAL) - builds a three character signal or equation name and places it in the simple equation string.
(BLANK) - skips to next non-blank character continuing over more than one card if necessary.
(OLDNEW) - writes simple equation out on disc or tape and resets counters for next equation.

This syntax applied to the equations in Figure 3 will first call the Semantic routine INITIL and then calls the line of syntax EQN. The first term of EQN will then skip the blanks before ABC. SIGNAL will read ABC and store it in the simple equation string, blanks will be skipped and the equals sign will be recognized. Again blanks will be skipped, NOW will be processed, blanks will be skipped and RSIDE will be called the character pointer at the first asterisk. RSIDE refers to OP which will recognize the asterisk and then DOG will be read and processed followed by recursion on RSIDE. Recursion will continue

LINES OF SYNTAX

(1)	LINE	=	(INITIL), EQN, RPT, (WRAPUP)
(2)	EQN	=	(BLANK), (SIGNAL), (BLANK), '=', (BLANK), (SIGNAL), (BLANK), RSIDE, (OLDNEW), ENDFNC
(3)	RSIDE	=	OP, (BLANK), (SIGNAL), (BLANK), RSIDE, OR, '@', ENDFNC
(4)	OP	=	'*', ENDFNC
(3A)	RSIDE	=	OP, (BLANK), (SIGNAL), (BLANK), RSIDE, OR, '@', OR, OP, (BLANK), OP, (TWOOP), OR, (SIGNAL), (TWOSIG), OR, OP, (BLANK), '@', (XTRAOP), ENDFNC
(4A)	OP	=	'*', OR, '+', OR, ', ', ENDFNC
(4B)	OP	=	'*', OR, '+', (STUCLA), OR, ', ', ENDFNC

LOGIC EQUATIONS

```
ABC = NOW * DOG * NL3 * NL6 * NBA

        * NFL   @   NL3=   CAT*DXE@

@@
```

- FIGURE 3 -

through the time NFL is read. The first term (OP) of RSIDE will now compare @ with * and go FALSE. The second alternative of RSIDE will recognize the @ and RSIDE will be TRUE. The syntax pointer will return to the term RSIDE in EQN and be incremented by one. OLDNEW will do its job and EQN will go TRUE. The syntax pointer will return to the second term of LINE, be incremented by one, and encounter the RPT. The pointer will be decremented one and EQN will process the next equation. After the last equation has been processed and EQN tries to process @@ it will be unable to do so and will go FALSE. The RPT will now be skipped and WRAPUP will be called terminating the scan. The @@ here served as an end of equations mark. Future reference will be made to this example.

SECTION 3

IMPLEMENTATION OF SYNTAX DIRECTED SCANNER-PROCESSOR

Number and Extent of Semantic Routines - One decision that has to be made by a programmer writing syntax is what extent should he break down tasks - how extensive should any one semantic routine be. At one extreme if we have a program DOIT, no matter how complicated, it can be trivially implemented in syntax via the syntax line: LINE = (DOIT). On the other hand hundreds or thousands of very special semantic routines could be utilized. We decided to let syntax do the job wherever possible, writing as many routines as need be. Then for tasks where the character scanning power of syntax would not be of value we decided to write semantic routines as large as necessary to do the job. Hybrid semantic routines were then written where two or more performed similar functions. The syntax was then rewritten to use these hybrids thus reducing the number of semantic routines.

This approach has worked well with the syntax currently calling about seventy semantic routines. (This includes routines for the CONTROLS processor not yet discussed.) The modularity implied by so many routines has frequently proved valuable when changes have been made. Negatively the effect on communication between routines in a related set is sometimes overlooked when changes are made to one routine in that set. This problem, however, has not proven to be serious.

PROBLEMS ENCOUNTERED

Two major problems were encountered while implementing our scanner-processor. The first of these was related to a particular convenience we allowed the logic designer. Normally some convention is employed when scanning statements for compilation or assembly to tell the scanner when the end of a statement has been reached. For instance a special character such as @ or $ may be required at the end of each statement. In FORTRAN column seventy three acts as an end-of-statement mark unless column six of the next card is non-blank. The first convention has the disadvantage that it is very easy to omit the special character at the end of each statement. The continuation character used by FORTRAN on successive cards of a long statement can also be forgotton but the fact that continuation cards are special makes such an omission less likely. However, the FORTRAN convention precludes placing more than one statement on a card. When the statements all begin in a distinctive and uniform manner, however, the end of one statement can be recognized by recognizing the beginning of the next. Since logic equations at Hughes all begin with a six character name followed possibly by blanks, and then an equals sign which is illegal in any other context it is quite straight forward to recognize a new statement. Thus, we are able to allow statements to extend over several cards or to allow more than one equation per card all without the use of continuation characters or end of statement marks. Unfortunately our first big problem arose when we tried to write syntax to process logic equations in this manner. Each of our approaches resulted in a recursion loop that required extensive stack sizes - on the order of several thousand locations each. Moreover, our attempts to combat this difficulty always seemed to hinge on forcing some end of statement convention on the logic designers. In essence we were recursing on each equation and for five thousand equations stack sizes were unrealistic. Eventually we resolved our difficulty by short-circuiting the stacks when recursion takes place on one line of syntax. If a term is the name of the line of syntax in which it occurs, then when the syntax pointer gets to this term the syntax pointer is set to point to the first term in the line of syntax as always but if the last entry in the stacks was also for recursion at this point this last entry will be replaced by the current pointers. Hence the number of entries in the stacks will not increase and no information is lost in the syntax pointer stack. This is because the last entry in the stack is identical with the current syntax pointer. For some applications the information

lost in the character pointer stack would create errors but for our application no error was introduced. As a result we now live harmoniously with pointer stacks accomodating only sixteen entries.

The second major problem we encountered was that of effective error flagging. Once we had solved our first problem, writing syntax to process error-free logic equations was relatively easy. In this context error-free means free from grammatical errors rather than free from logical errors. To be of maximum use to the logic designer the scanner-processor should do two things when it encounters an error. It should write a dianostic indicating the error and then it should skip the bad data continuing to look for errors in the remainder of the equations. Unfortunately it is a feature of syntax that unless the syntax has been written in such a manner as to anticipate the error, catastrophe results. Generally the syntax pointer will get lost and depending on the number and nature of checks in the syntax for other errors it may or may not ever find its way to a familiar spot. If it does the location and the nature of the error will likely be unclear. If it does not the attention of someone familiar with the syntax is then required and he may require several subsequent debug runs to find the error. Events of this type are received by a logic designer in a decidedly poor manner - especially if they occur more often than very infrequently. Consequently when writing syntax we try to anticipate errors a logic designer might make and then write the syntax so that these common errors will be recognized. Then the scanner-processor is able to issue a very explicit diagnostic and to set flags which the syntax can use to skip the error-containing equation. To illustrate consider again the example of Figure 3. Should an equation be written

```
NBA = NFL*CAB**BL3 @
```

when it is scanned RSIDE will go FALSE when it hits the two successive asterisks and hence EQN would be FALSE. Therefore the WRAPUP routine will be called terminating the scan. If on the other hand line of syntax (3) is replaced by line of syntax (3A) the error will be caught by alternative three of RSIDE and the semantic routine TWOOP can write a diagnostic pinpointing the error. Also TWOOP can put the character counter one past the next @ and return TRUE allowing the scan to continue with the next equation. Similarly, alternative four will recognize the error condition two signals in a row without a separating delimiter

```
NBA = NFL   CAB   @
```

and alternative five will recognize the condition that an extraneous delimeter appears after the last term

NBA = NFL * CAB * @

This alteration will of course not allow all errors to be recognized - the following would still abort the scan

NBA = NFL * CAB @ @

Still a significant improvement has been made and further modification could be made to handle this error as well. Clearly, however, as more complicated logic equations are allowed it becomes less possible and less desirable to handle all errors so cleanly. Therefore, in addition to explicit error traps set to catch particular errors our syntax is sprinkled with general error traps so that at least when an error is encountered, for which there is no explicit test, positive action can be taken. The scan can be prevented from aborting and a diagnostic telling approximately where the error occured can be written. The nature of the error will then have to be determined by logic designer inspection.

Changes to Syntax

Once our two major problems had been disposed of we found the use of syntax valuable. For instance a change in our approach to handling operators was very easily implemented by a simple change in the syntax and the addition of one short semantic subroutine. We had decided initially that we would not distinguish between the various operators (asterisk, plus, comma) when compiling the tables output by LOGCOM and hence had written the line of syntax

OP = '*', OR, '+', OR, ',', ENDFNC (line 4A Figure 3)

whose purpose was merely to see whether a legal operator appeared where one was expected. We felt that the logic designer could attach significance to those as his fancy dictated and we would treat them all the same - merely as delimeters. When it became necessary to actually distinguish plus (+) from the others the line of syntax above was rewritten as

OP = '*', OR, '+', (STUCLA), OR, ',', ENDFNC (line 4B Figure 3)

where the routine, STUCLA, inserts in the simple equation string the distinctive character string +UCLA+. This insertion is between the two terms separated by plus in the original equation. The equation

NF01/J = NF01/A * BF02/A + NF01/B

originally translated as

3 NF01/JNF01/A BF02/ANF01/B

is now translated as

3NF01/JNF01/ABF02/A+UCLA+NF01/B

This change was therefore, very easy to implement by making a simple change in the syntax. It should be mentioned that the insertion of +UCLA+ in the simple equation was a solution very readily handled by the table building subroutine.

SECTION 4

CONTROLS PROCESSOR

In addition to logic equations LOGCOM has numerous other inputs. All of those are read and processed by lines of syntax. These inputs fall into five categories known to LOGCOM as ALTERS (update), NAMECHANGE, TYPE, DEBUG and OPTIONS. A card is recognized as to category by which of the characters A, N, T, D or O it has in column 1. For instance, A indicates ALTERS, N indicates NAMECHANGE, etc. These control cards can be mixed together in any order except for the one restriction that all of the ALTER cards must be together. This restriction is necessary because it is desirable to process all of the ALTER's at one time. For the other categories, however, the cards are scanned and tables are built directed by syntax. Of these four, only the OPTIONS card will be discussed, although examples of the others are shown in Figure 4 to indicate the different formats.

Once the O in column one, signifying an OPTIONS card, has been recognized the scanner skips six characters so the full word OPTIONS may be written if desired. (Similar conventions exist for the other CONTROLS categories). It then scans the card looking for options

CONTROL CARDS

```
OPTIONS      NO MASTER      ,      LOGIC RELEASE
NAME         JT15/I = FT15/I
OPTIONS           INDEX, NPINS=16
ALTER        77, 79
ALTER        112, 112
   BA05DC  =  NB15CC        -  ENABLE
END ALTER
TYPE-N       NO.OUT  @   C=1  ,  D=2  @
TYPE-F       F=1  ,  T=2  @  C=1, D=2, P=3, S=4  @
OPTIONS         NO MERGED USED ON
NAME         JT14/I=FT14/I
```

- FIGURE 4 -

separated by commas. These options are written either in the form INDEX (NO INDEX) or NBOXES = 26. To each valid option corresponds a location in a status table when the status of each option is kept. For options that are either on or off, one and zero are used respectively to denote these states. For NBOXES = 26 the value 26 would actually be placed in the corresponding position in the table.

Syntax lends itself to this usage very well since the letters which spell an option such as INDEX can be recognized, in order, using literals. Then a simple semantic routine can set the correct core location to one -

'I','N','D','E','X',(SETONE)

Similarly

'N','O',(BLANK),'I','N','D','E','X',(SETZRO)

can set the correct core location to zero. Finally NBOXES = 26 can be handled

'N','B','O','X','E','S',(BLANK),'=',(BLANK),(SETINT)

where SETINT reads the 26, converts it to binary, and stores it.

A not immediately obvious advantage of such a method is the ability to define standard options. Standard values for each option can be internal data - assembled into the status table - and these values will be used unless changed by the contents of some OPTIONS card. This is convenient as usually the user only wishes to specify a few non-standard options. He can in fact elect all standard options and include no OPTIONS at all.

Moreover control cards in the other categories need only be present if information belonging to that category is to be read. Thus the user can obtain an index and set NPINS to 16 by just including data card three of Figure 4 anywhere in his data deck. This is considerably more convenient than placing a one in column twenty of data card two and sixteen in columns five and six of data card three. The non-standard options selected and the values set are clear from the data and the user need not worry about including dummy cards for sections of data he does not need to input. For instance in LOGCOM if he does not have NAME CHANGE data he does not have to include dummy data to tell LOGCOM no such data will be input this run. Also as indicated above the control cards are kept

together and in order.

As additonal options have been added to LOGCOM changing the syntax to handle them has been for the most part mechanical. This feature combined with the other advantages described has made the OPTIONS card a desirable feature for use by the other programs in our system. To date only one other program now possesses this capability but we expect to extend it to other programs as time permits.

SECTION 5

PROPOSED CHANGE TO LOGCOM

Logic designers at Hughes are currently writing equations for many circuits which are not satisfactorily represented by one AND-OR (NAND-NOR) equation. Flip-flops and adders are two such examples, and large scale integrated circuits can certainly be expected to be another. Our solution is to write several equations for such a circuit. For instance one equation will be written for each of the inputs of a flip-flop (clock, set, etc) or for each stage of an adder. In order to be handled by our partitioning program, however, these equations must satisfy nomenclature conventions which allow different equations describing the same circuit to be so recognized. An alternate procedure suggested by the Hughes logic design area is to write for these circuits pin-oriented equations. More specifically the right side of such an equation should consist of special terms separated by commas. A special term is a signal name enclosed in parentheses which is followed by a pin name. A sample equation would be:

MD32T. = (AF061T)B, (AF061S)C, (NL062T)E

If a one to one correspondence is established between pin names and positions in the simple equation string, the pin information will be present in the simple equation. The scanner-processor of LOGCOM reading the special term (AF061S)C would have to recognize the C and insert AF061S in the proper place in the simple equation. Moreover, when a pin is not used the corresponding positions on the simple equation would have to be left blank. The simple equation for the equation MD32T might then be:

4MD32T.AF061TAF061S NL062T

Changing the syntax so that it will also process equations of this type is not expected to be difficult and much of this work has already been done. At this point we are mostly waiting for clearer definition before we write and debug the remainder of the semantic routines.

SECTION 6

SUMMARY

To date our experiences with syntax have been gratifying. It has been effective both in the scanner-processor and CONTROLS sections of LOGCOM. The use of the OPTIONS card has been extended to another program and in the future its use should be extended to several more. In addition it has been proposed that syntax be written to read all or most of the data in other D A programs.

We have also written a syntax-directed program which resequences the external formula numbers in a FORTRAN deck. This resequencing is frequently useful for large decks. For us then syntax has been, and continues to be, a helpful and versatile tool.

APPENDIX A

BRIEF DESCRIPTION OF SYNTAX

Syntax is essentially a language whose major application is scanning and operating on character strings. A line of syntax is an instruction in this language which determines what actions will take place if specified tests are satisfied. Examples of lines of syntax appear in Figure 3 in the text. The name of a line of syntax is just the name appearing to the left of the equals sign. The right side of an equation consists of alternatives separated by OR's with the entire line terminated by ENDFNC. Alternatives consist of terms which are separated by commas. The individual terms fall into four categories:

1) literals which consist of a single character set off by apostrophes ('+');

2) names of semantic routines which are enclosed in parentheses;

3) names of other lines of syntax; and

4) the special terms OR, ENDFNC and RPT.

Associated with and necessary to the understanding of syntax are two pointers and the concepts of TRUE and FALSE. The syntax pointer keeps track of where we are in the syntax (which term of which line) and the character pointer points to that character in our data string which we are going to consider next. If the syntax pointer is pointing to a term β and the character pointer is pointing to a character α, the following will occur:

1) If β is a literal, α and β will be compared and the character pointer will be incremented by one. If α and β are equal a TRUE condition results which means that the syntax pointer is incremented by one. If they are not equal a FALSE condition results and the syntax pointer skips to the next OR or ENDFNC (whichever occurs first) unless β is immediately followed by a RPT. In this case the pointer is incremented by two so that it points to the term immediately after the RPT.

2) If β is the name of a semantic routine control will be transferred to this routine where some action will be performed. A semantic routine is just a subroutine with a recondite name which returns and is called in non-standard manners. It may read in characters (incrementing the character counters), make tests, build tables, call ordinary subroutines and eventually will terminate its activities returning either TRUE or FALSE (dependent on the success it enjoyed in its labors). The TRUE or FALSE condition that results has the same effect as when β was a literal. A TRUE condition causes the syntax pointer to be incremented and a FALSE condition causes the pointer to skip to the next OR or ENDFNC unless the next term is a RPT.

3) If β is the name of another line of syntax, the syntax pointer will be forced to point to the first term of that line of syntax. This is referred to as calling another line of syntax. Before the call, however, the current values of the syntax pointer and the character counter will be added to the syntax pointer stack and character pointer stack, respectively. These stacks are necessary for the special terms OR and ENDFNC.

4) If β is an OR or an ENDFNC and the last term was TRUE the entire statement in which β occurs is TRUE. Let us give this line the name RSIDE. The syntax and character pointers are then reset to the last values placed in the stacks and these values are deleted from the stacks. Evidently at some time in the past the syntax pointer was pointing to an occurrence of the term RSIDE which belongs to category three. (For instance term eight of EQN in Figure 3). At that time the syntax pointer was forced to the first term of the line RSIDE and now that RSIDE has been found TRUE the pointer is returning to that original term. Moreover since RSIDE is TRUE that term becomes TRUE and the syntax pointer will be incremented by one (to point to (OLDNEW) in EQN) and processing will continue.

If the last term tested was FALSE and β is an OR the syntax pointer is incremented by one and the next alternative will be processed. Also the character pointer will be reset to the value it had when processing begun on the preceding alternative. If β is an ENDFNC the pointers are reset from the last values put in the stacks as above but now the line RSIDE is FALSE. Therefore the term to which the syntax pointer is reset is FALSE and this FALSE term will have the same effect as always on the syntax pointer.

Finally, if β is a RPT and the preceding term was TRUE the syntax pointer is decremented by one and the preceding term is repeated. The effect of a RPT when the preceding term is FALSE has already been discussed. Essentially a RPT causes the preceding term to be repeated until it eventually goes FALSE at which time the syntax pointer skips to the term after the RPT.

Consider a sequence of lines of syntax, L_1, L_2, --- L_n, such that L_i calls L_{i+1}. Recursion is said to exist where a term in one of the lines, L_i, is the name of another of the lines, L_j, and $j \leq i$. In effect a loop is defined on the syntax. Recursion exists at term five of the line of syntax RSIDE in Figure 3.

To summarize a line of syntax consists of alternatives, separated by OR's, and terminated by an ENDFNC. An alternative is a string of terms separated by commas and each term is either a literal (category (1) above), the name of a semantic routine (2), the name of another line of syntax (3), or the special term RPT (4). Syntax and character pointers exist and refer to current locations. Stacks for these pointers exist and allow the syntax pointer to return to terms, which are names of lines of syntax (3), after the referenced lines have been determined TRUE or FALSE. Recursion is said to exist when a loop is defined in the syntax.