79-8

A Program that Paraphrases Variable-Valued Logic Formulas

J. Reiter

MLI 79-8

1979-8

A PROGRAM THAT

PARAPHRASES

VARIABLE-VALUED LOGIC FORMULAS

prepared by John Reiter

for

Professor R. S. Michalski

C. S. 490

Spring, 1979

1. PURPOSE OF THIS PAPER

This paper is intended to provide a brief description and serve as a user's manual for a program that can paraphrase variable valued logic formulas. Some implementation details will be provided; however, most of these details are decribed in the internal documentation of the program.

2. MOTIVATION FOR DEVELOPING THIS PROGRAM

2.1 USE IN KNOWLEGE-BASED AND COMPUTER INFERENCE SYSTEMS

The need to provide computer users with a simple, clearly understood interface to programs and to information systems has long been recognized. This need has been epecially apparent in the fields of knowledge base and computer inference system development.

The developers of the MYCIN medical knowledge base system show an awareness of this requirement in their design specifications. They wished to develop "a system capable of handling interactive dialog, and one which was not a 'black box'. This meant that it had to be capable of supplying <u>coherent</u> <u>explanations</u> of its results..."(1) The designers of MYCIN have also stated: "Augmentation or modification of any knowledge base is facilitated by the ability to discover what knowledge is currently in the system and how it is used. The sytem's acceptance (especially to a medical audience) will be strongly dependent upon the extent to which performance is <u>natural</u> (i.e. humanlike) and transparent". These requirements

have been met partly through the design of a program capable of supplying a paraprase of MYCIN production rules.

2.2 HISTORY AND GOALS

This program is based upon one which was developed for use in the MEDIKAS system, which is a system of programs being developed at the School of Basic Medical Sciences at the University of Illinois at Urbana-Champaign to assist in the construction and use of a medical knowledge base. That system uses slightly modified variable valued logic rules to direct the aquisition, modification, and display of information; therefore, the ability to paraphrase VL rules was essential in order to provide users (and the system developers) with clear, easily read versions of those rules.

The version described in this paper is a generalized and much more flexible version of the MEDIKAS paraphraser. It was written in a manner such that it can be easily easily adapted to a wide variety of programs that utilize variable valued logic formulas.

It achieves this adaptability by allowing the user to specify the format of the terminal symbols used in the formulas; for example, the user can specify the format of operators, function names, and variable names. The program also allows the user to specify the way in which the terminal symbols will be paraphrased. This will be explained in detail later in this paper.

3. PARAPHRASE EXAMPLES

3.1 DATA RULE

Before decribing the program in detail, several examples will be presented and explained (the examples can be found in figure 1).

The first example is a data rule extracted from the eastbound-westbound trains example used in "Pattern Recognition as Knowledge Guided Computer Induction" by R. S. Michalski. (reference 2). This example illustrates several features of the paraphrase. The paraphraser can print tokens just as they are received (such as "THERE EXISTS") or it can change the value of tokens (such as the change from "CAR1" to "THE FIRST CAR") or it can produce words that have no input token in that spot in the rule (such as the words "SUCH THAT"). It can also do some reordering, such as the paraphrase of "CONT-LOAD(CAR2,LOAD1)" to "THE SECOND CAR CONTAINS THE FIRST LOAD".

3.2 CLASSIFICATION RULE

This example also demonstrates all of the features mentioned in the previous section.

3.3 RULE WITH INTERNAL OPERATORS

This rule demonstrates that internal conjunction and internal disjunction operators are recognized and can be paraphrased in a reasonable manner.

```
THERE EXISTS CAR1,CAR2,CAR3,CAR4,LOAD1,LOAD2
[INFRONT(CAR1,CAR2)]&[INFRONT(CAR2,CAR3)]&[LENGTH(CAR1)=LONG]&
[CAR-SHAPE(CAR1)=ENGINE]&[CAR-SHAPE(CAR2)=U-SHAPED]&[CONT-LOAD(CAR2,LOAD1)]&
[LOAD-SHAPE(LOAD1)=TRIANGLE]&[NRWHEELS(CAR3)=2] ::> [CLASS=EASTBOUND].
```

```
THERE EXISTS THE FIRST CAR AND THE SECOND CAR AND THE THIRD CAR AND
THE FOURTH CAR AND THE FIRST LOAD AND THE SECOND LOAD SUCH THAT IF
THE FIRST CAR IS IN FRONT OF THE SECOND CAR AND THE SECOND CAR
IS IN FRONT OF THE THIRD CAR AND THE LENGTH OF THE FIRST CAR IS LONG
AND THE SHAPE OF THE FIRST CAR IS AN ENGINE AND THE SHAPE OF
THE SECOND CAR IS U-SHAPED AND THE SECOND CAR CONTAINS THE FIRST LOAD
AND THE SHAPE OF THE FIRST LOAD IS A TRIANGLE AND THE NUMBER
OF WHEELS ON THE THIRD CAR IS 2 THEN THE CLASSIFICATION OF THE TRAIN
IS EASTBOUND .
```

```
THERE EXISTS CAR1,CAR2,LOAD1,LOAD2 [INFRONT(CAR1,CAR2)]&[CONT-LOAD(CAR1,LOAD1)]&
    [CONT-LOAD(CAR2,LOAD2)]&[LOAD-SHAPE(LOAD1)=TRIANGLE] &
     [LOAD-SHAPE(LOAD2)=POLYGON] ::> [CLASS = EASTBOUND].
```

```
THERE EXISTS THE FIRST CAR AND THE SECOND CAR AND THE FIRST LOAD AND
THE SECOND LOAD SUCH THAT IF THE FIRST CAR IS IN FRONT OF
THE SECOND CAR AND THE FIRST CAR CONTAINS THE FIRST LOAD AND
THE SECOND CAR CONTAINS THE SECOND LOAD AND THE SHAPE OF
THE FIRST LOAD IS A TRIANGLE AND THE SHAPE OF THE SECOND LOAD IS
A POLYGON THEN THE CLASSIFICATION OF THE TRAIN IS EASTBOUND .
```

```
THERE EXISTS CAR1, CAR2 [CAR-SHAPE(CAR1) . CAR-SHAPE(CAR2) =
ENGINE, JAGGED-TOP . "U-SHAPED"].
$
```

```
THERE EXISTS THE FIRST CAR AND THE SECOND CAR SUCH THAT THE SHAPE OF
THE FIRST CAR AND THE SHAPE OF THE SECOND CAR IS AN ENGINE OR
JAGGED-TOPPED OR U-SHAPED .
```

Figure 1.

4. OVERVIEW OF THE PROGRAM

4.1 GENERAL INFORMATION

The program has been written on the CDC CYBER 175 in the standard version of Pascal as described in Pascal User Manual and Report by Kathleen Jensen and Niklaus Wirth (reference 3). It contains approximately 1500 lines of code and comments and requires 75K of memory to compile.

The program was written to be reasonably efficient; however, great emphasis was not placed on speed efficiency, since the only real requirement is that the paraphrase be produced in real time. The program performs well within this bound.

The program also does not use a large amount of main memory, which is important if this program is to be integrated with other programs. There are some space utilization improvements that can be done; these are described in section 8 of this paper.

4.2 LEXICAL ANALYSIS

The first major part of the program is the lexical analyzer, which determines if the current input string starts with one of the types of tokens which the parser expects at that point.

The lexical analyzer is rule driven; that is, it interprets the rules which the user supplies to indicate the format of the fourty terminals in the grammar. A Backus-Naur Form (BNF) description of the grammar for these terminal specification rules follows (nonterminals are typed in lowercase, terminals are underlined, and alternate options are specified by the "!" character):

```
specification ::= ALPHA ! DIGIT !
     ' specification '
     [ specification ] ! / specification / !
     specification ! specification !
     specification , specification !
     ( specification )
```

ALPHA specifies one alphabetic character, DIGIT specifies one digit, 'characters' specifies a string consisting of those printable characters (for example, 'CAR' specifies the string consisting of the three letters C, A, and R), [specification] specifies repetition of specification zero or more times, / specification / specifies one or more repetitions, ! between specifications means that both specifications are allowed, a comma between specifications means concatenation (for example, 'CAR',/DIGIT/ means the string CAR followed by one or more digits). Parentheses can be used to override the default implicit parenthesization, which is right parenthesization (that is, 'CAR'!'LOAD',/DIGIT/ means 'CAR'!('LOAD',/DIGIT/)). Appendix A contains the terminal specification rules used to generate the examples of figure 1. The rules must be typed in the order shown.

4.3 PARSING

A top-down, recursive parser is used to parse formulas, producing an explicit parse tree as it does so. The grammar that is parsed reflects VL21 syntax as specified in "Pattern Recognition as Knowledge-guided Computer Induction" (reference 2). A BNF description of the grammar that is accepted by the parser can be found in Appendix B.

Note that this grammar is quite general; for example, constants, variables, and functions are accepted as arguments to functions. Also, note that constants can (but need not be) in quotes; therefore, ambiguity arising from the possible confusion between functions and constants as references can be handled by placing quotes around the constants, but if no confusion is possible, quotes are not needed.

One thing to note about the parser is that it tells the lexical analyzer which token to look for, therefore simplifying the lexical analyzer's job. Most lexical analyzers do there job independently of the parser, so that it must decide out of all possible types of terminals into which class the current one fits.

The advantage to the user of using the type of lexical analyzer that the paraphraser uses is that it gives the user greater freedom in choosing the format of different terminals; for example, the symbol chosen for one of the selector operators could be the same as one chosen for one of the metaimplication operators and no confusion would arise (since at no point does the parser expect both a selector operator and a metaimplication operator to be possible valid input).

For each grammatical contruction that the parser recognizes, some binary tree representation of that structure is produced. Illustrations showing the structure of the tree for each terminal in the grammar are presented in Appendix C.

An understanding of these tree structures is essential in order to specify the paraphrase rules that are described in the next section. Understanding most of these structures is not difficult, however. Binary operators such as the selector operators are placed in a node with its operands as its left and right subtrees. List separators such as argument list separators are considered to be binary operators with the parts of the list that it separates as operands. Unary operators such as NOT have only one non-nil subtree (the right subtree). Functions are are considered to be unary operators with their argument list as their operand. Nullary functions such as variables and constants have no non-nil subtrees.

The parse tree that was contructed for the paraphrase of the third example in figure 1 is shown in figure 2. This example shows each of the types of subtrees described in the previous paragraph.

4.4 PARAPHRASING..

After the parser (with the help of the lexical analyzer) has successfully parsed the formula and produced a binary parse tree, a routine is called upon to traverse the tree and produce a paraphrase as it does so. This routine functions by interpreting user-supplied rules that specify the format of the paraphrase to be performed for each terminal; or, to be

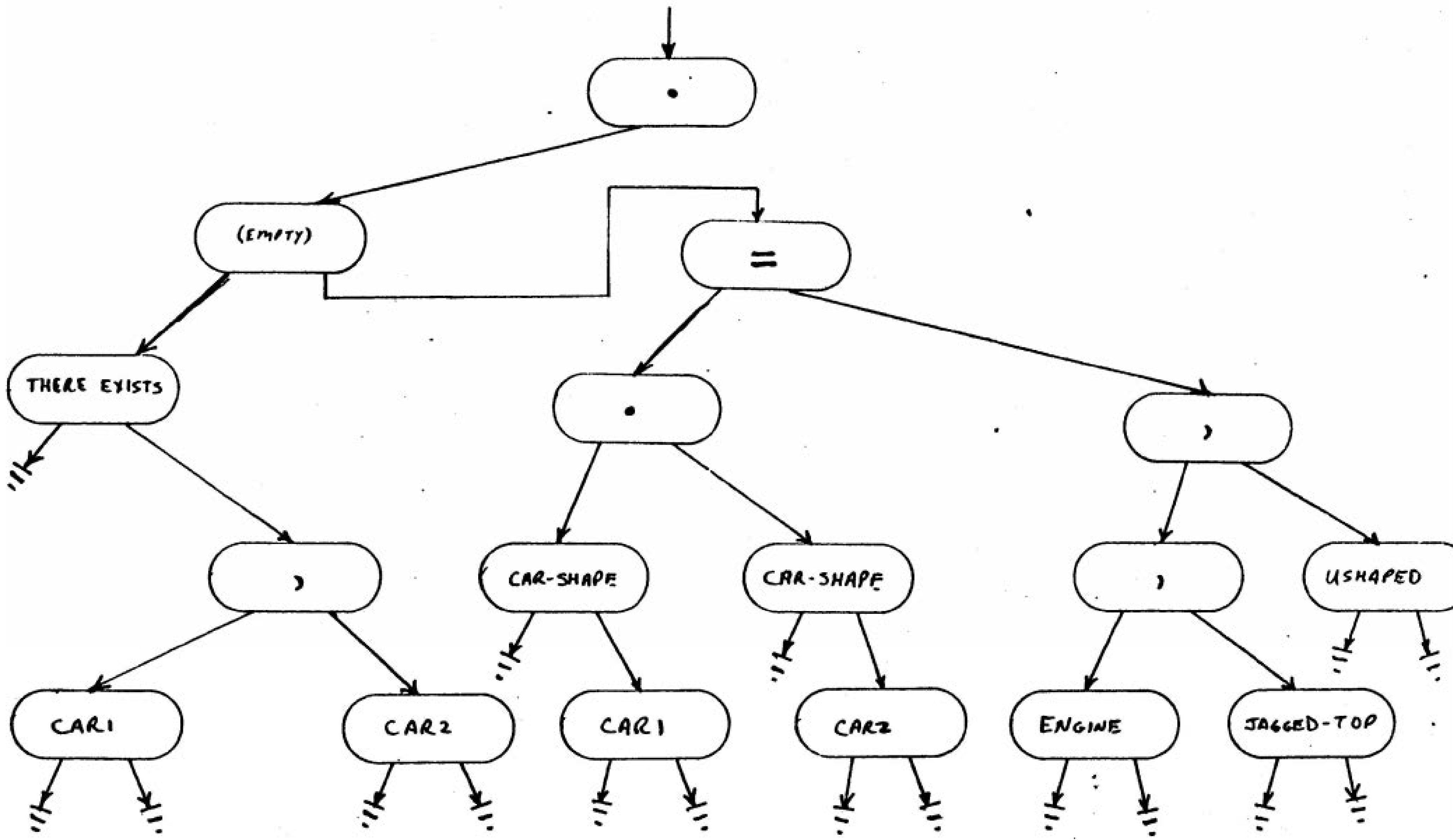


Figure 2.

more specific, the rule for each terminal specifies how to paraphrase the subtree of which that terminal is the root node.

A BNF description of the grammar of the paraphrase rules is:

```
paraphraserule := inputspecification :

     outputspecification .

inputspecification ::= terminalidentifier  !

     terminalidentifier = ' valueofterminal '

terminalidentifier ::= characters(except : =)  ! "


outputspecification ::= empty  !  outputitem  !

     outputspecification outputitem

outputitem ::= ' characters '  !  V  !

     leftorright P

leftorright ::= L ! R  !  leftorright L  !
     leftorright R
```

The rule interpreter attempts to match the input specification part of the rules for the current token being paraphrased. Note that there are two formats for input specifications. The first format consists of only

a terminal identifier, which is a character string which labels the corresponding terminal or is a " (which means that the label of the last rule to have a character string applies to the current rule). For example, FUNCTIONNAME could be the terminal identifier of a rule and " could be the terminal identifier of the following rule (indicating another function name rule). The second format of input specifications allows the specification of a value for the terminal; for example, FUNCTIONNAME = 'INFRONT' will match only the function whose value is INFRONT.

The output specification part can be empty, in which case no output is produced for that input specification. If it is V, then the value of the terminal is printed out. If it is a string of printable characters enclosed in quotes, then that string is printed out. If it is P preceded by a series of one or more R's or L's (R meaning traverse right, L meaning traverse left, and P meaning paraphrase), then the paraphrase routine will recursively call itself to paraphrase the part of the tree specified by the L's and the R's. For example, RP says to paraphrase the right subtree, and RLP says to paraphrase the subtree obtained by traversing right and then left from the current node. Specifications such as RLP and RRP are especially useful for paraphasing function arguments.

A listing of the rules used to produce the paraphrases shown in figure 1 can be found in Appendix D. They illustrate all of the features decribed in this section. Note that the terminal specifications are actually just comments as far as the paraphrase rule interpreter is concerned, thus a quote <u>must</u> be used to indicate a another rule for the preceding non-quote terminal specification. The order of the rules must be as shown.

Also note that the rule interpreter checks the rules for the current token in the order in which they are given; therefore, more than one rule may potentially match the current token but only the first rule that matches is used. This feature is useful for specifying an "else" rule. For example, the last CONSTANT rule in Appendix D is

```
" : V.
```

which is matched for any constants (such as numbers) which do not match previous rules. This allows one to bother to write a special rule for only those cases which need special paraphrasing.

5. RUNNING THE PROGRAM

All programs and data files are presently in the account with user number 3KNPU30. The source is in the file named PARA, the terminal rules in Appendix A are in TRULES2, the variable valued logic rules are in VRULES2, and the paraphrase rules are in PRULES2. To compile the source program, use the following commands (follow all GET commands with /UN=3KNPU30):

```
GET,PARA
GRAB,PASCAL
RFL,75000
PASCAL,PARA,LIST
```

To run the program, type

```
GET,TRULES2
GET,VRULES2
GET,PRULES2
LGO,TRULES2,VRULES2,PRULES2,RESULTS
```

This will place the paraphrase in the file named RESULTS. Using OUTPUT instead of RESULTS should print the output to the screen.

6. PROPOSED CHANGES AND EXTENSIONS

6.1 EXTENSIONS

One nice feature that should be added to the input specification part of the paraphrase rules is some method of checking the right or left subtree for the occurrence of some input specification. This feature would allow the selector operator '=' to be paraphrased as 'is' when the referee is singular and 'are' when it is plural. Also, by being able to check for nil subtrees, a list of values like CAR1, CAR2, CAR3 could be paraphrased as CAR1, CAR2 and CAR3' as is done in English. See figure 1 if it is not apparent that this could be done with this feature added.

Another extension would be to allow some terminals to have empty value in some cases (such as "and" between selectors). Allowing any terminal to be empty in all cases invites ambiguity problems, however.

6.2 IMPLEMENTATION IMPROVEMENTS

Several things could be improved in the program. First, the format of the terminal specification rules could be made more flexible. Presently, no blanks are allowed in the rules. Also, each rule should have some way of specifying which terminal it is defining.

Better detection of errors in the rules used by the program would be desirable. Recovery from some errors would also be desirable.

In addition, more clever, space efficient data structures could be used to save some of the space used for rule storage.

7. BIBLIOGRAPHY

(1) Davis, R., Buchanan, B. and Shortliffe, E., "Production Rules as a Representation for a Knowledge-Based Consultation Program", Stanford Artificial Intelligence Laboratory Memo AIM-266, October 1975.

(2) Michalski, R. S., "Pattern Recognition as Knowledge-Guided Computer Induction", Department of Computer Science, University of Illinois at Urbana-Champaign Report No. UIUIDCS-R-78-927, June 1978.

(3) Jensen, Kathleen, and Wirth, Niklaus, "Pascal User Manual and Report", Springer-verlag, 1974.

APPENDIX A

TYPE OF TERMINAL	DESCRIPTION
...RMULA TERMINATOR	'.'
...ONST. LIST SEPARATOR	':'
...ARIABLE	('CAR'./DIGIT/)!('LOAD'./DIGIT/)!'CLASS'!'NRCARS'!'NRCARS-LENGTH-L[illegible]
...R. LIST SEPARATOR	','
...OR ALL	'FOR ALL'
...ERE EXISTS	'THERE EXISTS'
EXCEPT	'\'
...MPLICATION	'->'
...UIVALENCE	'<->'
...ISION ASSIGN.	'!!>'
...FERENCE	'=>'
...ENERALIZATION	'!<'
...EMANT. EQUIV.	'!='
...OR	'OR'!'V'
...L. OR	'EOR'
...ND	'&'!'AND'!'Λ'
...R. LEFT CLOSE	'('
...R. RIGHT CLOSE	')'
...OT	'NOT'
TRUE	'TRUE'
FALSE	'FALSE'
...NKNOWN	'?'
...ECTOR LEFT CLOSE	'['
...ECTOR RIGHT CLOSE	']'
... CONST.	'.'
...UCTION NAMES	'INFRONT'!'LENGTH'!'CAR-SHAPE'!'CONT-LOAD'!'LOAD-SHAPE'!'NRPTS-LOAD'!'NRWHEELS'!'NR-PTS-LOAD'!'POSITION'
EQUALS	'='
NOT EQUALS	'<>'
...REATER THAN	'>'
...ESS THAN	'<'
...T. OR EQUALS	'>='
...T. OR EQUALS	'<='
...NT. DISJ.	','
...ANGE OPERATOR	':'
...EFT QUOTE	'"'
CONSTANT	'ENGINE'!'U-SHAPED'!'T-[illegible]'!'POLYGON'!'OPEN-TOP'!'JAGGED-TOP'!'L[illegible]'EASTBOUND'!'WESTBOUND'!/DIGIT/
...IGHT QUOTE	'"'
...LIST. LEFT CLOSE	'{'
...LIST RIGHT CLOSE	'}'
...LIST SEPARATOR	','

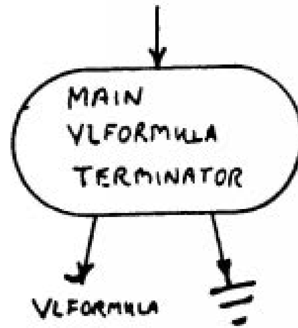
APPENDIX B

BNF DESCRIPTION OF TYPICAL VARIABLE VALUED LOGIC FORMULA SYNTAX

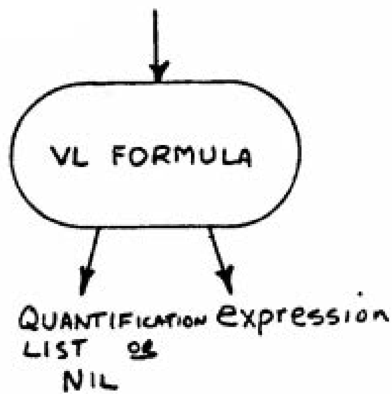
```
<main formula> ::= <vlformula> <MAIN FORMULA TERMINATOR>

<vlformula> ::= <quantification list> <expression> !
        <expression>

<quantification list> ::= <quantification> !
        <quantification list> <QUANT LIST SEPARATOR> <quantification>

<quantification> ::= <quantifier> <variable list>

<variable list> ::= <VARIABLE> !
        <variable list> <VARIABLE LIST SEPARATOR> <variable>

<quantifier> ::=  <FOR ALL> ! <THERE EXISTS>

<expression> ::= <subexpression> !
        <expression> <metaoperator> <subexpression>

<subexpression> ::= <product> !
        <subexpression> <EXCEPT> <product>

<metaoperator> ::= <IMPLICATION> ! <EQUIVALENCE> ! <DECISION ASSIGNMENT> !
        <INFERENCE> ! <GENERALIZATION> ! <SEMANTIC EQUIVALENCE>

<product> ::= <factor> !
        <product> <or operator> <factor>

<or operator> ::= <INCLUSIVE OR> ! <EXCLUSIVE OR>

<factor> ::= <term> ! <factor> <AND> <term>

<term> ::= <EXPRESSION LEFT CLOSE> <expression> <EXPRESSION RIGHT CLOSE> !
        <NOT> <expression> !
        <TRUE> ! <FALSE> ! <UNKNOWN> !
        <selector> ! <vlformula>

<selector> ::= <SELECTOR LEFT CLOSE> <referee> <selector op> <reference>
        <SELECTOR RIGHT CLOSE> !
        <SELECTOR LEFT CLOSE> <referee> <SELECTOR RIGHT CLOSE>

<referee> ::= <atomic function> !
        <referee> <INTERNAL CONJUNCTION> <atomic function>

<atomic function> ::= <FUNCTION NAME> !
        <FUNCTION NAME> <argument list> ! <VARIABLE> .

<selector op> ::= <EQUAL> ! <NOT EQUAL> ! <GREATER THAN> ! <LESS THAN> !
        <GREATER THAN OR EQUAL> ! <LESS THAN OR EQUAL>

<reference> ::= <reference item> !
        <reference> <INTERNAL DISJUNCTION> <reference item>

<reference item> ::= <extended function> !
        <extended function> <RANGE OPERATOR> <extended function>

<extended function> ::= <FUNCTION NAME> !
        <FUNCTION NAME> <argument list> ! <VARIABLE> !
        <LEFT QUOTE> <CONSTANT> <RIGHT QUOTE> ! <CONSTANT>

<argument list> ::= <ARG LIST LEFT CLOSE> <arguments> <ARG LIST RIGHT CLOSE>

<arguments> ::= <extended function> !
        <arguments> <ARG SEPARATOR> <extended function>
```

APPENDIX C

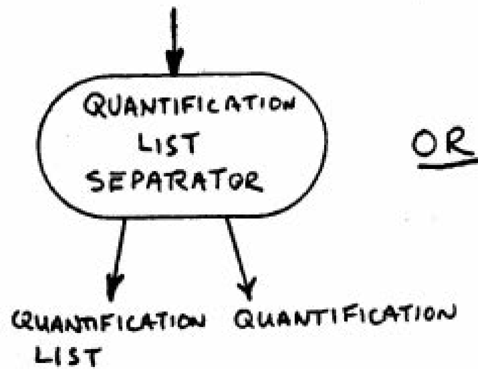
MAIN VL FORMULA:



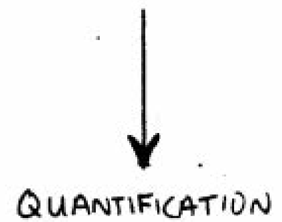
VL FORMULA:



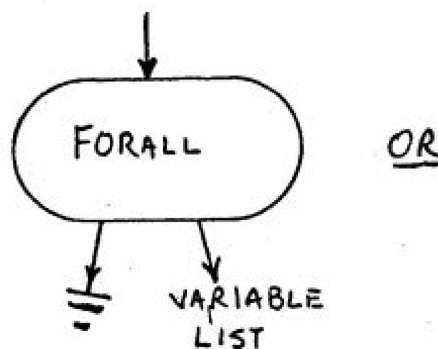
QUANTIFICATION LIST:



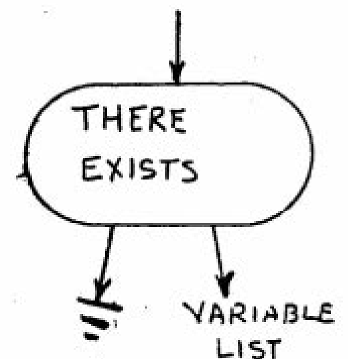
OR



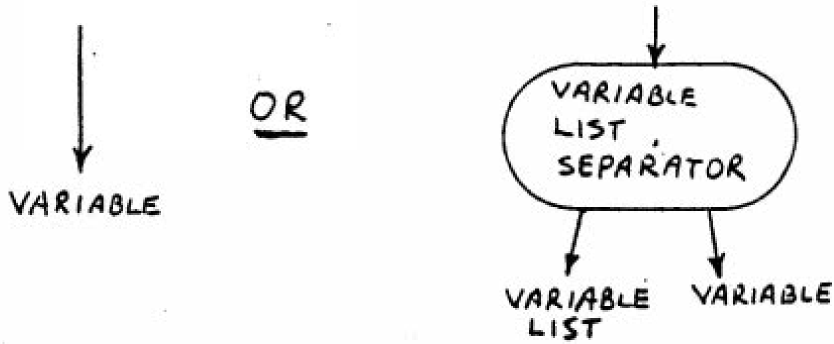
QUANTIFICATION:



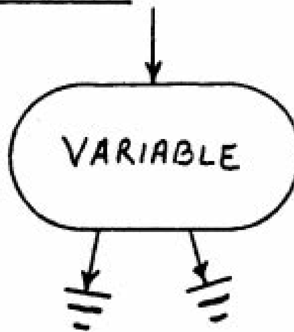
OR



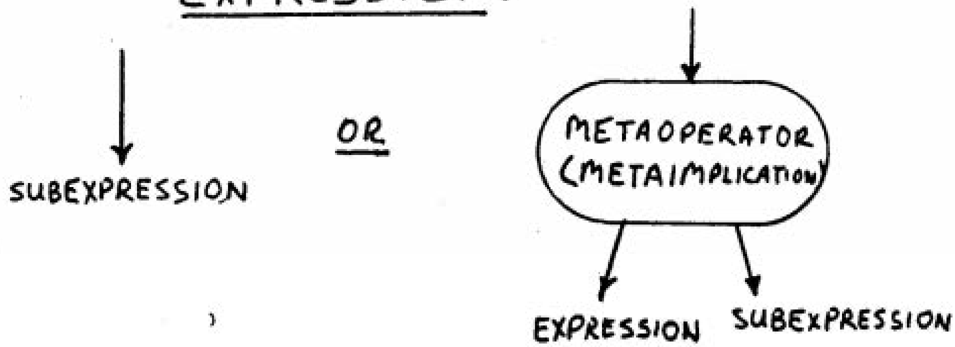
VARIABLE LIST:



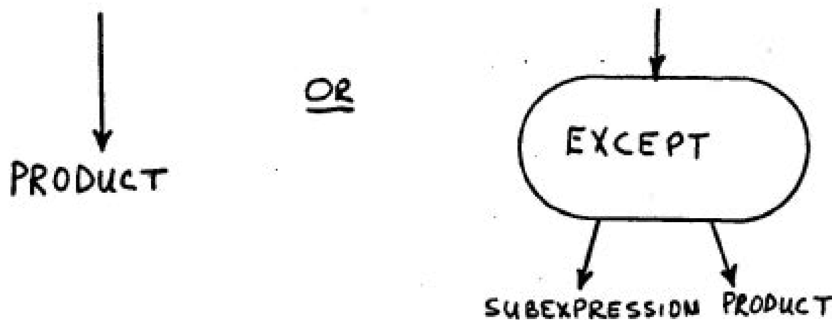
VARIABLE:



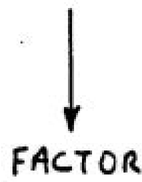
EXPRESSION:



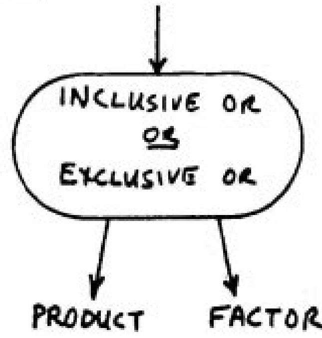
SUBEXPRESSION:



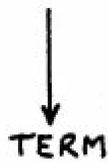
PRODUCT



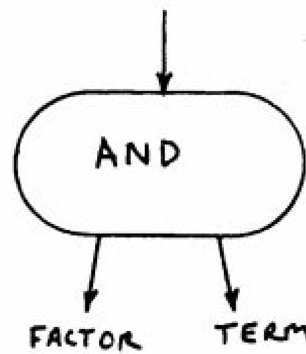
OR



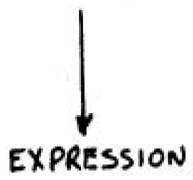
FACTOR



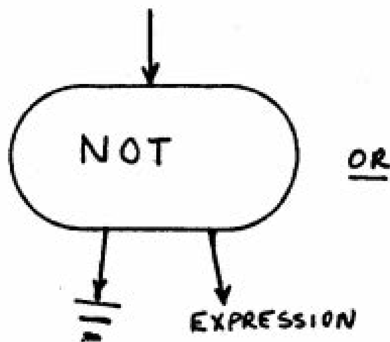
OR



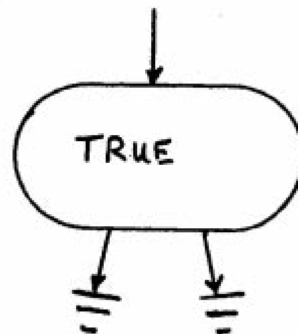
TERM



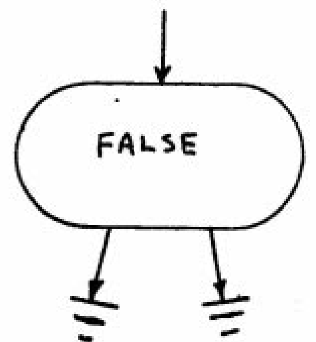
OR

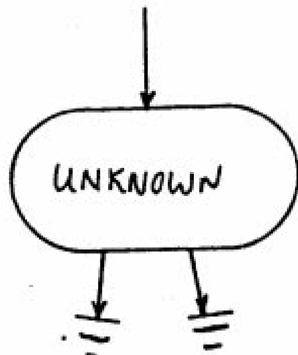


OR

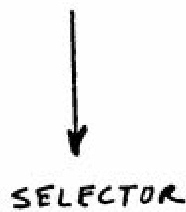


OR

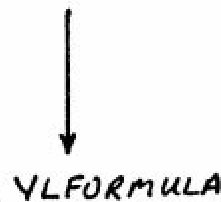




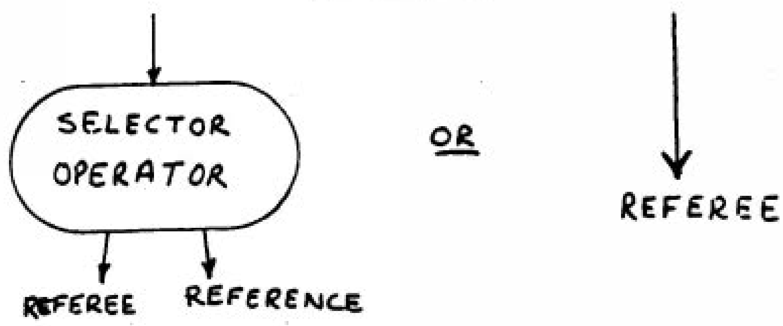
OR



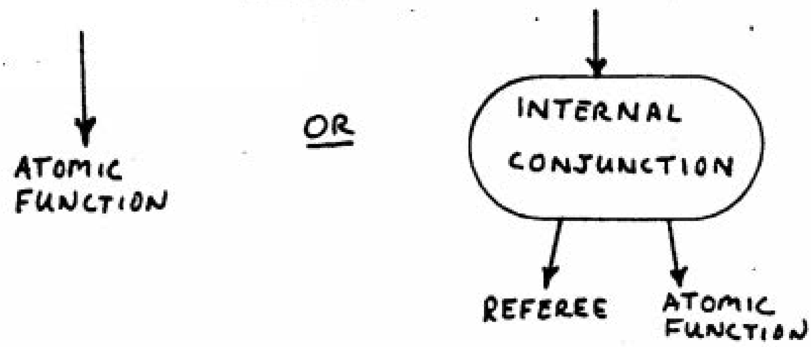
OR



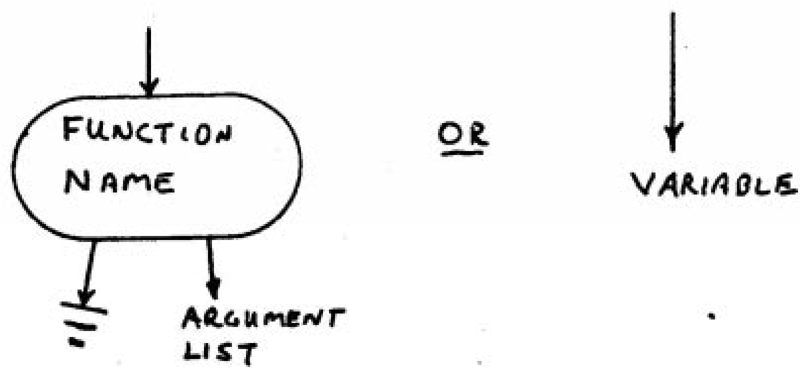
SELECTOR



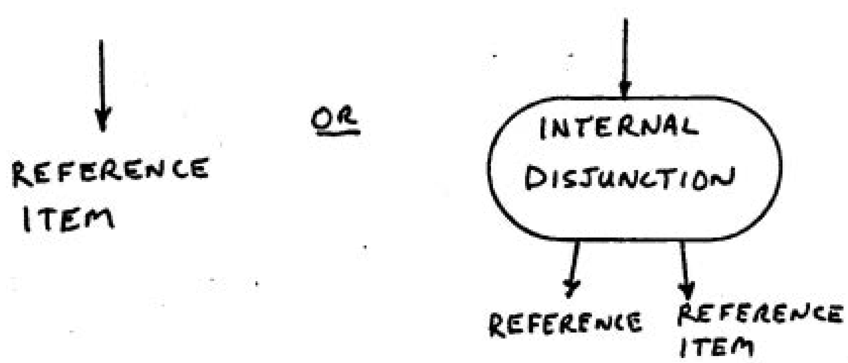
REFEREE



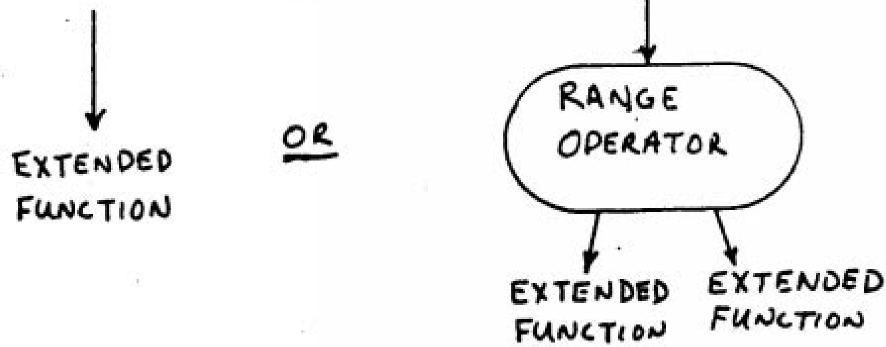
ATOMIC FUNCTION



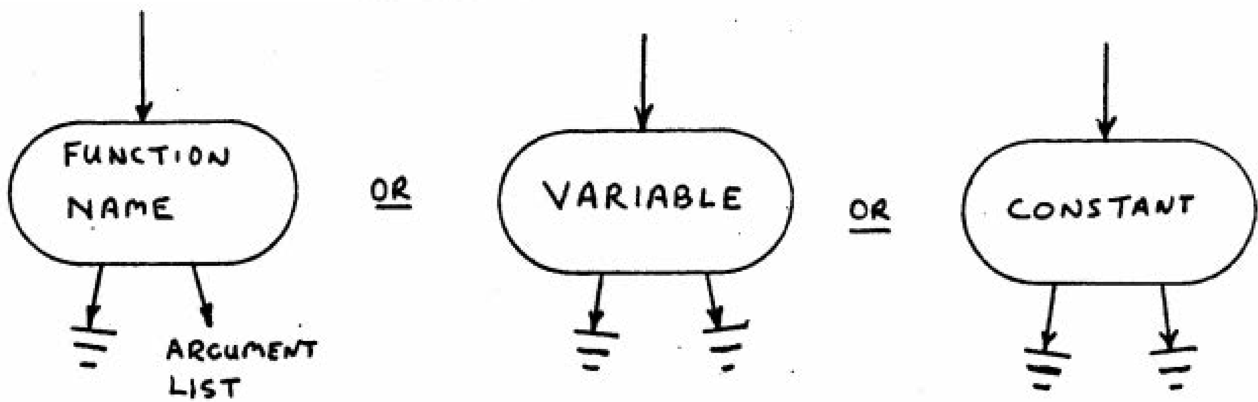
REFERENCE



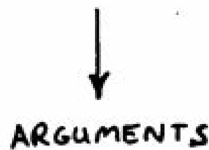
REFERENCE ITEM



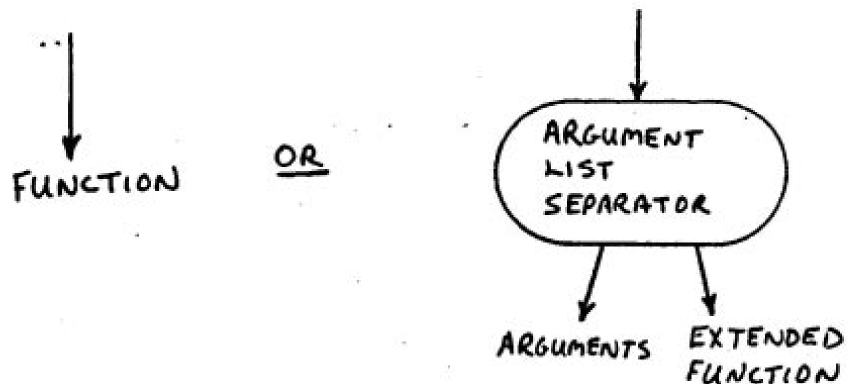
EXTENDED FUNCTION



ARGUMENT LIST



ARGUMENTS



APPENDIX D

```
VLFORMULA : LP RP.
VLFORMULATERMINATOR : LP '.'.
QUANTLISTSEPARATOR : LP ',' RP.
VARIABLE = 'CAR1' : 'THE FIRST CAR'.
   "     = 'CAR2' : 'THE SECOND CAR'.
   "     = 'CAR3' : 'THE THIRD CAR'.
   "     = 'CAR4' : 'THE FOURTH CAR'.
   "     = 'CAR5' : 'THE FIFTH CAR'.
   "     = 'LOAD1': 'THE FIRST LOAD'.
   "     = 'LOAD2': 'THE SECOND LOAD'.
   "     = 'NRCARS' : 'THE NUMBER OF CARS' 'IN THE TRAIN'.
   "     = 'NRCARS-LENGTH-LONG' : 'THE NUMBER' 'OF LONG CARS'.
   "     = 'CLASS' : 'THE CLASSIFICATION' 'OF THE TRAIN'.
   "     : V.
VARLISTSEPARATOR : LP 'AND' RP.
FORALL : 'FOR ALL' RP 'SUCH THAT'.
THEREEXISTS : 'THERE EXISTS' RP 'SUCH THAT'.
EXCEPT : LP 'EXCEPT' RP.
IMPLICATION : 'IF' LP 'THEN' RP.
EQUIVALENCE : LP 'EQUALS' RP.
DECISIONASSIGNMENT : 'IF' LP 'THEN' RP.
INFERENCE : LP 'INFERS THAT' RP.
GENERALIZATION : LP 'GENERALIZES TO' RP.
SEMANTICEQUIVALENCE : 'THE FACT THAT' LP ' SEMANTICALLY EQUALS' RP.
INCLUSIVEOR : LP 'OR' RP.
EXCLUSIVEOR : LP 'OR BUT NOT BOTH' RP.
AND : LP 'AND' RP.
EXPRLEFTCLOSE :.
EXPRRIGHTCLOSE :.
NOT : 'NOT' RP.
TRUE : 'TRUE'.
FALSE : 'FALSE'.
UNKNOWN : 'UNKNOWN'.
SELECTORLEFTCLOSE :.
SELECTORRIGHTCLOSE :.
INTERNALCONJUNCTION : LP 'AND' RP.
FUNCTIONNAME = 'INFRONT' : RLP 'IS IN FRONT OF' RRP.
     "       = 'LENGTH' : 'THE LENGTH OF' RP.
     "       = 'CAR-SHAPE' : 'THE SHAPE OF' RP.
     "       = 'CONT-LOAD' : RLP 'CONTAINS' RRP.
     "       = 'LOAD-SHAPE': 'THE SHAPE OF' RP.
     "       = 'NRPTS-LOAD': 'THE NUMBER OF PARTS' 'IN THE LOAD OF' RP.
     "       = 'NRWHEELS' : 'THE NUMBER' 'OF WHEELS ON' RP.
     "       = 'POSITION' : 'THE POSITION OF' RP.
     "       : V RP.
EQ : LP 'IS' RP.
NE : LP 'IS NOT' RP.
GT : LP 'IS GREATER THAN' RP.
LT : LP 'IS LESS THAN' RP.
GE : LP 'IS GREATER THAN OR EQUAL TO' RP.
LE : LP 'IS LESS THAN OR EQUAL TO' RP.
INTERNALDISJUNCTION : LP 'OR' RP.
RANGEOPERATOR : 'IS BETWEEN' LP 'AND' RP.
LEFTQUOTE :.
CONSTANT = 'ENGINE' : 'AN ENGINE'.
   "     = 'TRIANGLE' : 'A TRIANGLE'.
   "     = 'POLYGON' : 'A POLYGON'.
   "     = 'OPEN-TOP' : 'OPEN-TOPPED'.
   "     = 'JAGGED-TOP': 'JAGGED-TOPPED'.
   "     : V.
RIGHTQUOTE :.
ARGLISTLEFTCLOSE :.
ARGLISTRIGHTCLOSE :.
ARGLISTSEPARATOR : ','.
```