THE DESIGN AND IMPLEMENTATION OF A STRUCTURED PROGRAMMING LANGUAGE WITH FEW ARBITRARY RESTRICTIONS - THE COMPILATION PHASE

A dissertation presented by

NOLA M. SIMPSON

In partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

at

Massey University

June, 1973

ABSTRACT

This thesis outlines the design and implementation of a structured teaching language with particular emphasis on the compilation phase. This language, which has been called MUSSEL (Massey University Structured Student Language) is intended for instruction of first year Computer Science students. In this respect it is a language which is as free as possible from arbitrary syntactic restrictions and is in a form that the student should find both easy and natural to write, with a minimum incidence of programming errors.

It is evident that the language a student uses profoundly influences the way he develops his techniques and habits of construction of algorithms. MUSSEL has been designed with this influence in mind and has been deliberately designed as a structured language i.e. the language reflects the basic structure of programs, so that algorithms can be refined and expressed naturally in it. In this sense MUSSEL aims to teach the student programming as a constructive intellectual discipline rather than just as a tool to communicate with the computer.

MUSSEL has been implemented as an interpretive system i.e. during the compilation phase the source language is translated into an intermediate language, which, is then interpreted. The reasons influencing this type of implementation are the need in a teaching language to minimise compile time and to maximise diagnostics, both at compile-time and at execution-time.

ACKNOWLEDGEMENTS

In being able to present this thesis, I wish to offer sincere thanks to the following people.

To my supervisors, Professor G. Tate and R.W. Doran for their help, guidance and forbearance.

To Peter Gibbons, with whom I worked for many long hours in discussing the design and implementation of MUSSEL.

To the Mathematics Secretary, Mrs. I.M. Cornish for her kindness, tolerance and competence in typing the script.

Finally to the Staff of both the Computer Unit and Mathematics Department for their constant reminders during the past six months, that my thesis was due for completion.

I would also like to acknowledge the fact that changing my name has not been exactly conducive to completing this thesis within the respective time limit.

Massey University
June, 1973.

Nola M. Simpson

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION - A STRUCTURED LANGUAGE FOR STRUCTURED PROGRAMMING.

It has been common practice when introducing programming to students for the first time, to furnish them with the details and idiosyncrasies of a given programming language, mostly FORTRAN or BASIC, and then by means of examples attempt to teach students how to write programs in the particular language.

As a result it is not surprising if the student after this course is hardly an effective programmer, and often has not even been taught a basis on which to further develop his capabilities. After all, he has only been subjected to what Wirth [6] has called a "computer appreciation course" or maybe a "disappreciation course"

It should be recognised that programming is a more difficult subject than is normally estimated and that the real art of programming is firstly construction of algorithms, and this has in the past normally been left to the intuition of the student alone. Increasing the complexity of the problems used as examples, merely increases the demands and strain on the students intuition, but not further his capability to control and check his intuition by reason - he may even be misguided into believing that the essence of programming is concocting ad-hoc ideas and tricks to solve given problems.

The only effective solution to this dilemma is to extend the education in programming to that of construction of algorithms.

Up to the present time there exists neither an established theory on this subject or any well-based methods, as these are still in the process of development.

In an attempt to overcome this failure in programming education Dijkstra [3] and later Wirth [6] have proposed a structured approach to programming and algorithm development. Their method is in effect a process of analysing an algorithm into smaller and smaller independent component parts until it has been broken down into basic elementary instructions. Dijkstra used this approach initially to prove the correctness of programming, but has since advocated its use as a general method and maintains programming efficiency is stepped up by a significant factor.

Wirth [8] developed his own language PASCAL, and in a paper [6] describes how he uses it in the structured approach to programming. PASCAL is a very highly structured language the sequencing being expressed without goto statements and it also deals with structured data. In his paper Wirth shows how to develop in parallel a program and the structure of the data on which it is to operate.

Using his approach, program development can be achieved provided a teacher had at his disposal any suitable structured language, but, unfortunately many teachers do not have such a language available, or if they do have one e.g. PL/1, Algol, it may not be possible to use it because the compiler is too slow or the error diagnostics are bad for a beginning student.

Up to the present time, this has been the situation at Massey University where the only student language available has been FORTRAN. FORTRAN of course is an unstructured language i.e. the sequence of execution of basic instructions by the computer has to be laid out explicitly with goto or similar statements. For describing programs written in an unstructured language such as FORTRAN the flow diagram has been widely used as it is also explicitly structured and so is very suitable for this purpose.

In attempting to teach structured programming to students Doran [10], [11], [12] has devised what he has called "Structure Diagrams" , as a notation to replace flow diagrams for the description of algorithms. The basic idea behind producing a structure diagram is to successively divide an algorithm into smaller and smaller subalgorithms. The notation used, follows the structure of Wirth's language PASCAL, and in fact closely resembles the parse trees in his language. The process of successively dividing an algorithm has been called by Dijkstra "refining an algorithm". In a structure diagram this refinement of an algorithm is represented pictorially by a "tree", the nodes of which are boxes essentially containing statements of subalgorithms. Since the algorithms represented may be recursive these diagrams are really 'lists' not 'trees'.

This method of formulating algorithms has proved to be very effective, but there was no language available, which was suitable or compatible with these ideas. Algol is a structured language, although not sufficiently so, and also it is not the easiest language to introduce to beginning students.

To overcome this difficulty it was decided to design a new language called MUSSEL, which would be compatible with the ideas on structured programming and yet simple enough for the average student to be able to grasp it quickly and naturally. Of course, a new language should not be developed just for the sake of novelty and existing languages should be used as a basis for development wherever possible. In this sense this has been true in the design of MUSSEL and PL/1 has been used as the basis since it was the language which met most of the requirements to a higher degree than most other languages.

To summarise, MUSSEL has been designed and implemented with the following principal objectives in view:

(i) To design a language with the structure and syntax to resemble very closely the form of Doran's Structure Diagrams, so that programs written in the language may be expressed in a systematic, precise and appropriate way.

(ii) To design a language which will enable programming to be taught from basic concepts which are clearly and naturally reflected in the language.

(iii) To make a language available with a syntax which is as natural as possible for a student to use, and also with as few arbitrary restrictions as possible.

(iv) To illustrate that a language with a rich set of program structuring facilities can be implemented on a small computer and be reasonably efficient for student use.

(v) To provide a framework, which due to its systematic structure, can be at a later date adapted to the various needs for which it has been designed, for example a compiler system for first year computer science students, with a large set of error checking facilities and diagnostics and the ability to gather a wide range of user statistics.

We believe that in the design of MUSSEL the first three of our objectives have been achieved, and thus we hope, using MUSSEL a teacher could shift the emphasis in teaching programming from that of learning the details of a language, to that of the construction and development of algorithms, which is where the emphasis should be. In the implementation of MUSSEL although hampered by the computer available at the present time, we trust the remaining two objectives will be achieved.

CHAPTER 2

A DESCRIPTION OF THE MUSSEL LANGUAGE

2.1 Admissible characters

MUSSEL uses a basic character set consisting of the 26 upper case letters of the alphabet A through Z, the digits 0 through 9, and the special characters +-*/!:.,b()$.

2.2 Constants

There are 3 types of constants in MUSSEL.

(i) Numbers

These may be either real or integer, the latter being written without a decimal point or exponent.

examples: 5 -3000 278.3496 +1.0 -6.5E-7 1E10 .00463E44

All numbers are represented internally by MUSSEL as a 7 digit mantissa (with the implied decimal point to the right) accompanied by a 2 digit exponent. Under this representation, integers are treated essentially as a subset of the reals and are distinguished internally by a zero exponent. The range of integers therefore is from -9999999 to +9999999, while reals may range from $\pm$ 1E-99 to $\pm$ 9999999E99.

(ii) Booleans

These may be written as TRUE or FALSE, or in the abbreviated forms T or F. Internally they are represented by the digits 1 and 0 respectively.

(iii) Strings

A string in MUSSEL consists of zero or more characters from the basic character set, excluding the record mark. Constants are enclosed between exclamation (!) marks. Similar to the convention adopted in PL/I, each exclamation mark itself in a string is represented by 2 such marks.

examples: !THIS IS A STRING!
!! (null string)
!!!! (single exclamation mark)
!!!QUOTE!!! (represents !QUOTE!)

2.3 Operators

The 4 types of operators used in MUSSEL expressions are as follows:

(i) Arithmetic

These include +, -, *, /, ./. (integer divide), and ** (exponentiation).

(ii) Logical

Used within conditional expressions, these include .AND., .OR., and .NOT.

(iii) Relational

Written as in FORTRAN IV, these include .LT., .LE., .EQ., .NE., .GE., and .GT., being equivalent to the relational operators $< \leq \neq = \geq >$ as used in ALGOL.

(iv) String

Strings may be concatenated by the use of the operator .CAT. Other common string manipulations may be performed by the use of library functions.

The above operators have the following precedence:

Operator	Priority
.OR.	1
.AND.	2
.NOT.	3
.LT. .LE. .EQ. .NE. .GE. .GT.	4
.CAT.	5
+ -	6
* / ./. NEG	7
**	8

(NEG indicates unary minus)

2.4 Names

These are used in MUSSEL to identify variables, labels, procedures and functions. Although a name clearly should be as short as possible, in many languages it is often hard to find a good mnemonic to satisfy the restricted length requirements. For this reason MUSSEL has imposed no such restrictions, allowing names to be of any length, consisting of a letter followed by a string of letters and/or digits.

examples: I SUM U124R902 VERYVERYVERYLONGNAME

2.5 Variables

All main variables used in a MUSSEL program must be reserved at the head of the program.

example: RESERVE X,SUM,MEAN,VARIANCE

The form of this instruction indicates clearly to students the process of setting aside storage space for variables. Another nice feature for the student is the absence, unlike FORTRAN, PL/I, ALGOL and COBOL, of explicit type declarations. Each variable's type is set implicitly through an assignment instruction and hence does not remain static throughout a program. The implementation of this is not a great drawback as MUSSEL, being an interpretive system, can easily check the types of variables during the execution phase.

2.6 Arrays and subscripted variables

Arrays in MUSSEL may be 1 or 2 dimensional only, which although restrictive is considered adequate for student requirements. Although the system does provide for dynamic arrays (see procedures), main arrays, like outer block arrays in ALGOL, must have constant bounds.

example: RESERVE A(1:60,2:10), B(-1:1), (C,D,E(-4:4,0:5))

Subscripts in MUSSEL, like ALGOL and PL/I, may be any arithmetic expression with the convention that non-integral values are rounded to the nearest integer.

examples: A(I), B(J,K), C(A(4), 2*SIN(4*PI*B(-4,0)))

2.7 Assignment Instructions

It is unfortunate that most languages, including FORTRAN and PL/I, use the equal (=) sign to specify the assignment operation.
example: I=I+1

The criticism here of course is that the meaning of this sign contrasts with its meaning in algebra. FORTRAN has the further disadvantage of not catering for multiple assignments.

To emphasise the assymmetry of the assignment relationship, ALGOL in fact replaces the equal sign by ':=', but even here the new operator resembles closely the equal sign. A further criticism of ALGOL is that the multiple assignment, although evaluated quite unambiguously according to the rules in the ALGOL report, is in a form which may cause trouble to students.

For these reasons MUSSEL has an assignment instruction of the form

SET...loc...TO exp

Analogous to the MOVE verb format in COBOL, this form expresses multiple assignments in a natural way and emphasizes quite clearly the operation of replacing the contents of a location (or locations) by the value of an expression.

examples: SET I TO I+1
SET A,B,C TO 0
SET DATE TO DAY.CAT.MONTH.CAT.YEAR

2.8 Program structuring

Apart from procedures, which are discussed below, program structuring in MUSSEL is achieved through the use of instruction groups. These are equivalent to compound statements in ALGOL and similar to DO groups in PL/I i.e. they are optionally labelled sets of instructions grouped between the instruction parentheses DO and END, each group being treated syntactically as a single instruction. The absence (without procedures) of a more complex structure, such as the block in ALGOL, simplifies the structuring of MUSSEL a great deal. All main variables are reserved at the head of a program and are, together with labels, global to the whole of that program. In this way a student is able to structure and display his program clearly (indenting 5 columns per group) without having to master the more involved variable scope problem present in ALGOL.

2.9 Conditional Groups

A variety of conditional instruction groups is provided in MUSSEL. The IF-group, of the form

(i) IF condexp THEN simple instruction

or (ii) IF condexp
THEN instruction

corresponds to similar IF-THEN statements in ALGOL and PL/I. FORTRAN IV's logical IF statement corresponds to form (i) only.

examples:
```
IF X.GT.Y .OR. A.LT.B THEN SET F TO 1
IF N .GE. M
THEN DO
        SET MEAN TO SUM/M
        EXIT
    END
```

MUSSEL's conditional group, of the form

```
DO IF condexp
THEN instruction
ELSE instruction
END
```

corresponds to similar IF-THEN-ELSE statements in ALGOL, PL/I and COBOL, while the rather unnatural arithmetic IF statement is its counterpart in FORTRAN.

example:
```
DO IF I .EQ. 0
THEN DO
        SET S TO N+K
        SET P TO P+S
     END
ELSE DO
        SET S TO N-K
        SET P TO P-S
     END
END
```

A feature of MUSSEL which is unavailable in the 4 mentioned languages is the general choice group. The use of this feature avoids the complicated nesting of conditional groups in some programs which would thereby obscure the simple structure of the program.

Example:
```
DO CHOICE OF
     IF TEMP .LT. 0  THEN SET STATE TO !FREEZING!!!
     IF TEMP .LT. 10 THEN SET STATE TO !COLD!
     IF TEMP .LT. 15 THEN SET STATE TO !COOL!
     IF TEMP .LT. 25 THEN SET STATE TO !MILD!
     IF TEMP .LT. 30 THEN SET STATE TO !WARM!
     IF TEMP .LT. 50 THEN SET STATE TO !HOT!
     ELSE SET STATE TO !BOILING!!!
END
```

The 'ELSE...' in the above form is optional.

Another feature is the CASE group which has its counterpart in Burroughs' ALGOL. This is MUSSEL's equivalent of the computed GO TO in FORTRAN, PL/I and COBOL, and the switch in ALGOL. Its use may be seen from the following simple decoding example:

```
DO CASE N OF
     SET STUDENT TO !FRESHER!
     SET STUDENT TO !2ND YEAR!
     SET STUDENT TO !3RD YEAR!
     SET STUDENT TO !4TH YEAR!
END
```

An advantage in MUSSEL however is that by the insertion of bounds, the index need not necessarily range from 1.

example:
```
DO CASE OPCODE IN (22,26) OF
     SET OP TO !SUBTRACT!
     SET OP TO !MULTIPLY!
     SET OP TO !COMPARE!
     SET OP TO !TRANSMIT DIGIT!
     SET OP TO !TRANSMIT FIELD!
END
```

2.10 Loops

The repeated group, of the form

```
DO REPEAT (control)
     ...instruction...
END
```

provides for extensive looping facilities in MUSSEL. The variety of controls available correspond essentially to the PERFORM verb available in COBOL.

examples: (i)
```
DO REPEAT FOR I FROM FIRST TO LAST BY STEP
     ...
END
```

(ii)
```
DO REPEAT FOR STRING SET TO !IT'S!,!AN!,!EXAMPLE!
     ...
END
```

The forms displayed above essentially correspond to those available in ALGOL (with its 'for' statement) and PL/I (with its 'DO variable = iteration list' statement), with the additional freedom of allowing set list elements in form (ii) to be of any type. (Note - FORTRAN's DO statement corresponds to form (i) above with the restriction that all controlling variables be positive at run time). As is allowed in these 3 languages, the STEP in form (i) may be omitted if 1. Additionally MUSSEL allows the test value also to be omitted.

examples: (iii)
```
DO REPEAT FOR I FROM 12*K**2
     ...
END
```
(iv)
```
DO REPEAT FOR I FROM 1 TO 100
     ...
END
```
(v)
```
DO REPEAT FOR I FROM N./.4 BY 2*C(U,V)
     ...
END
```

As in PL/I, the controlling expressions in the above forms (i), (iii), (iv) and (v) are all called by value rather than by name if the loop is regarded as a procedure i.e. the expressions are evaluated and the results stacked on entry to the loop at run time and are thus unaffected by any changes within the loop. This both simplifies the implementation and avoids some nasty features of ALGOL.

MUSSEL also provides for iteration without indexing a variable.

```
examples:   (vi)  DO REPEAT
                      READ X
                      IF X .EQ. 9999 THEN EXIT
                      SET SUM TO SUM+X
                  END
           (vii)  DO REPEAT UNTIL P .EQ. Q
                      ...
                  END
          (viii)  DO REPEAT WHILE X .NE. 0
                      ...
                  END
```

The last form only is available in PL/I.

Another feature of MUSSEL, unavailable in ALGOL, PL/I, or FORTRAN, but available using the PERFORM verb in COBOL, is the 'REPEAT valexp TIMES' loop.

```
example:   DO REPEAT N TIMES
               READ X
               SET SUM TO SUM+X
               SET SUMSQUARES TO SUMSQUARES+X*X
           END
```

Where REPEAT groups are nested, the word EXIT specifies an exit from the innermost loop. If an exit from another loop is required, then the appropriate label must be specified also.

```
example:   OUTLOOP: DO REPEAT FOR I FROM 1 TO M
                        DO REPEAT FOR J FROM 1 TO N
                            IF B(I,J) .EQ. 0
                            THEN DO
                                     SET STATE TO !ERROR!
                                     EXIT FROM OUTLOOP
                                 END
                            SET A(I,J) TO A(I,J)/B(I,J)
                        END
                    END OUTLOOP
```

It is important to note that MUSSEL has no equivalent of the GO TO statement of other languages. The reason for the absence of this unconditional branch instruction is to preserve the structure of a program. In this way a student is encouraged to refine his program achieving transfer of control by means of conditional groups and EXIT instructions, or through the use of procedures and functions.

2.11 Procedures

As there are no explicit type declarations in MUSSEL, procedures and functions have similar declaration forms. These must be placed at the head of a program, immediately following the main variable reservations, and are of the following general form:-

```
DEFINE procedure name (ON formal parameter list) AS
DO
    (RESERVE local variable list)
     ...instruction...
END
```

example:

```
DEFINE SWAP ON A,B AS
DO
     RESERVE HOLD
     SET HOLD TO A
     SET A TO B
     SET B TO HOLD
END
```

A function value assignment in MUSSEL is accomplished by use of the VALUE IS instruction.

example:

```
VALUE IS X**N/FACTORIAL(N)
```

The form of this instruction indicates to students the special nature of this assignment.

The MUSSEL translator distinguishes procedures and functions by their manner of call. A procedure call is specified by the EXECUTE statement.

examples: EXECUTE SWAP(A,B)
EXECUTE TRANSPOSE(A,N)
EXECUTE P

A function call on the other hand is indicated by the appearance of the function name (and actual parameters, if any) in an expression, as in most other languages.

examples: SET NIBBLE TO SUBSTRING(WORD,1,4)
SET Y TO F(A**2,B*C)

Parameters in MUSSEL may be expressions or procedure names and are called by reference, as in PL/I. In particular, whole arrays may be altered by passing across array names as parameters to procedures. This has the additional advantage that space in a procedure need only be reserved for the array word address, rather than the array elements themselves.

As in ALGOL, the scope of variables reserved within a procedure does not extend beyond that procedure. However, variables reserved in the main program are global to all procedures, and any procedure may call any other, including itself. There is just one restriction - procedure declarations may not be nested. The purpose of this is to preserve the simple structure of the language. The block structure is essentially 2-level, as opposed to the multi-level structure of ALGOL which may be difficult for the novice programmer to comprehend.

2.12 Recursion

It is well known that recursion plays a powerful and essential role in the processing of many data structures. There are many areas (e.g. list processing) in which to program iteratively, rather than recursively, would be quite awkward, if not impossible in practice. Yet a student learning FORTRAN for example, which does not allow a subroutine to be called recursively, is forced to think and program iteratively, and is thereby not encouraged to look for recursion. This may become a bad habit - when introduced to the concept at a later date, he may find it hard to understand.

For these reasons, recursion is considered an essential facility in MUSSEL. By programming in the language, a student may be introduced to the recursive way of thinking at an early stage and will thereby be in a position to weigh the respective merits of iterative and recursive programming.

2.13 Dynamic Arrays

These are reserved locally to a procedure. Bound expressions must be arithmetic, consisting of main variables and/or parameters to the procedure. As is the case with subscript expressions, the convention is that non-integral bound values are rounded to the nearest integer.

examples: RESERVE A(1:N),(B,C(BOT:TOP)),D(0:1*J/K,L:E(U,V))

These rules are similar to those applying to bound expressions in ALGOL. In FORTRAN IV however, where dynamic arrays are permitted in subprograms, subscripts and bounds must be (restricted) integer expressions ranging up from 1.

2.14 Input/Output

In FORTRAN input/output, in particular the FORMAT statement, has long been a major bug for students. For example, a recent analysis of 511 first year student programs at Massey revealed that in fact nearly 30% of all programming errors were linked with the formatting of I/O. Yet this is an area, we believe, which should not concern the student unduly at an early stage in his programming career - initially there are more fundamental concepts for him to grasp. For this reason MUSSEL has endeavoured not only to simplify I/O as much as possible for the novice, but also to allow him at a later stage to format his output in a clear and natural manner.

2.14.1 Input

MUSSEL has a simple unformatted stream input with data being separated on cards by a blank or a comma. String constants have the additional delimiter '!'. Arrays are read in row-wise by the specification of the array name.

```
example:   RES X,Y(1:10),Z(0:19,0:49)
           READ X,Y,Z
```

2.14.2 Output

For the novice programmer, MUSSEL provides a standard formatted output. Here again arrays are output row-wise by specifying the array name in the PRINT list.

example: PRINT X,Y,Z

At a later stage however the student may optionally include a PICTURE specification with his output data, similar to the forms used in COBOL and PL/I. Although this may appear to be a primitive form of specification, we believe it will aid the student in visualising exactly how his O/P will appear on the printed page.

Characters used in the MUSSEL picture specification are as follows:-

* : The position indicated will contain an alphanumeric character. With a number, however, leading zeros will be automatically suppressed unless otherwise specified.

B : A blank is inserted in the position indicated.

Those used specifically with numeric data are as follows:-

. : A decimal point will be printed in the position indicated.

, : A comma is printed in this position if there is a printed digit to the left; otherwise a blank is printed.

9 : A zero will not be suppressed in this position.

S : If this character is placed at the head of a picture, the sign (+ or -) will be floated up to and printed at the head of the number concerned. Otherwise the sign of the number will be printed in the position indicated.

- : Similar to S, except that the sign is printed only if the number is less than zero. Otherwise a blank is printed.

E : The character 'E' is printed in the position indicated. This is used in the E - format specification of a number. Note that leading exponent zeros are automatically suppressed unless otherwise specified.

In addition the following rules apply:-

(i) Replication of the characters '*', 'B', and '9' may be specified by a bracketted integer following the character concerned.

example: *(4).*(2) means ****.**

(ii) The association of a picture with a variable remains optional, the output being in standard format if a picture is omitted.

(iii) Unlike COBOL, the picture accompanies the variable name concerned in a PRINT list, and is of the form (PIC= picture).

example: PRINT A,B(PIC=***.**),C(PIC=S***.**ES99)

(iv) Whole arrays may be output by specifying the array name and a single picture only. Each element is then printed out according to the picture with a standard 5 spaces between elements.

example:
```
RES A(0:M,0:N)
...
PRINT A(PIC=****.***)
```

(v) Character strings are left justified (or left adjusted) under a PICTURE specification i.e. if necessary a string is extended with blanks on the right or truncated on the right to the specified field width.

examples:

Internal String	PIC=	Output
HEADING	*(7)	HEADING
HEADING	*(10)	HEADINGbbb
HEADING	*(4)	HEAD

(vi) The required output form of Boolean constants may be specified by picture. If a picture is absent, the form TRUEb or FALSE is printed.

examples: Boolean Value	PIC=	Output
TRUE	none	TRUEb
FALSE	none	FALSE
TRUE	*	T
FALSE	*	F
TRUE	****	TRUE
FALSE	****	FALS
TRUE	*(10)	TRUEbbbbbb
FALSE	*(10)	FALSEbbbbb

(vii) With pictures specifying numeric data no more than one of each of the characters . and E is allowed per picture.

examples:

Internal Representation		PIC=	Output
Mantissa	Exponent		
0001234	00	****	1234
		********	bbbb1234
		S*****.*ES99	bb+123.4E+01
		**999999	bb001234
-0012345	-03	****.***S	bb12.345-
		***.*	b12.3
		****.*E-**	1234.5E-b2

It can be seen that the PIC specification used in MUSSEL differs in a number of ways from that used in COBOL and PL/I. The main reason for these changes and simplifications are as follows:-

(i) As MUSSEL is intended primarily as a student language, it is felt that a cheque protection character (e.g. * in COBOL) would not be used often enough to justify its inclusion in the picture specification. This leaves the asterisk as a reasonable character to use to specify an alphameric position in a picture. If however it should be required as a cheque character, then this could quite easily be programmed into the output by use of string processing subroutines. In a similar way the $ character, although not considered necessary as a special character, could quite easily be appended to output if necessary.

(ii) MUSSEL uses the picture specification to edit output only. The character 'V', specifying an implied decimal point, is therefore not required.

(iii) We believe it is more natural to have leading zeros in a number automatically suppressed unless otherwise specified.

(iv) We believe it is important that a picture correspond as closely as possible to the intended printed O/P. For this reason insertion of the sign of a number is not automatically implied in MUSSEL. A student using the language must specifically ask for the sign to be appended by the use of S or - and if this is not done, then the sign will be omitted.

(v) MUSSEL does not, as in COBOL, use the picture specification to define a variable's type.

2.14.3 Printer Controls

The following are used in MUSSEL:-

```
NEWPAGE      - skip to newpage
NEWLINE      - skip to newline
NEWLINE(N)   - skip N lines
SPACE        - shift one space to right
SPACE(N)     - shift N spaces to right
TAB(N)       - shift to (N+1)th. column
```

```
example:  PRINT NEWPAGE,TAB(30),!HEADING!,NEWLINE,TAB(30),!*******!,
               NEWLINE(2),TAB(28),!X= !X(PIC=***.**)
```

The output with X containing 339.56 is as follows:-

```
... 30 spaces ...       HEADING
                        *******
... 28 spaces ...     X= 339.56
```

2.15 Conclusion

What can one say about the design of MUSSEL in comparison with the other four languages FORTRAN, Algol, PL/1 and COBOL?

(i) It is more structured than any of the above languages. There is no provision in MUSSEL for unconditional transfer statements akin to the goto's in the other languages. This means algorithms have to be refined down to the basic statements. In this respect it can be compared with Wirth's highly structured language PASCAL [8], which only has a certain type of instruction for terminating loops, similar to MUSSEL.
It can be said, with justification that where a student learns to program in a language which has provision for unconditional transfer statements, he is more than likely going to make excessive use of such statements, thus hiding the meaning of his program and at the same time teaching him bad programming habits. (Dijkstra [4] has stated that a programmers efficiency can be measured inversely to the number of gotos he uses. It has been proved that students, who are taught without this facility, never miss them and learn good programming habits from the start.

(ii) MUSSEL has no explicit type declarations as the other languages have. This is a good feature in a teaching language, because students should not be unduly worried with such matters when learning to program initially. Wirth's language EULER is also a typeless language and like MUSSEL has been implemented as an interpretive system. The elimination of type declaration is not a serious drawback in a interpretive system and it could be said that type declarations appear in other languages as a facility to aid the compiler writer rather than as aid to the programmer.

2.15 cont'd

(iii) The Input/Output facilities have been kept simple and unsophisticated with the addition of a much simpler PICTURE specification than either COBOL or PL/1. Languages that have been designed specifically for teaching purposes, such as CORC [25], have taken into account the difficulty beginning students experience in getting their data read into the computer and producing their results in a readable form, but, they have made no provision for students to have at their disposal any added facilities, so that they may dictate how their printed results are to appear. By introducing the PICTURE into MUSSEL, we have given them this provision and it will be beneficial to them later when they start to program in other languages.

(iv) MUSSEL, through its REPEAT groups and general CHOICE groups, provides a richer set of program structuring than is provided for in FORTRAN, Algol, PL/1 or COBOL, and in this respect compares more than favourably with PASCAL.

2.16 AN EXAMPLE

The following example (a simplified version of that used in Henderson and Snowdon [13]) illustrates just how easily a MUSSEL program follows from a structure diagram.

Problem

A program is required to process a stream of telegrams. This stream is available as a sequence of words. Each telegram is delimited by the word !ZZZZ!. The stream of telegrams itself is terminated by the occurrence of the empty telegram i.e. the null word!!. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words !ZZZZ! and !STOP! are not chargeable and words of more than 12 letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word.

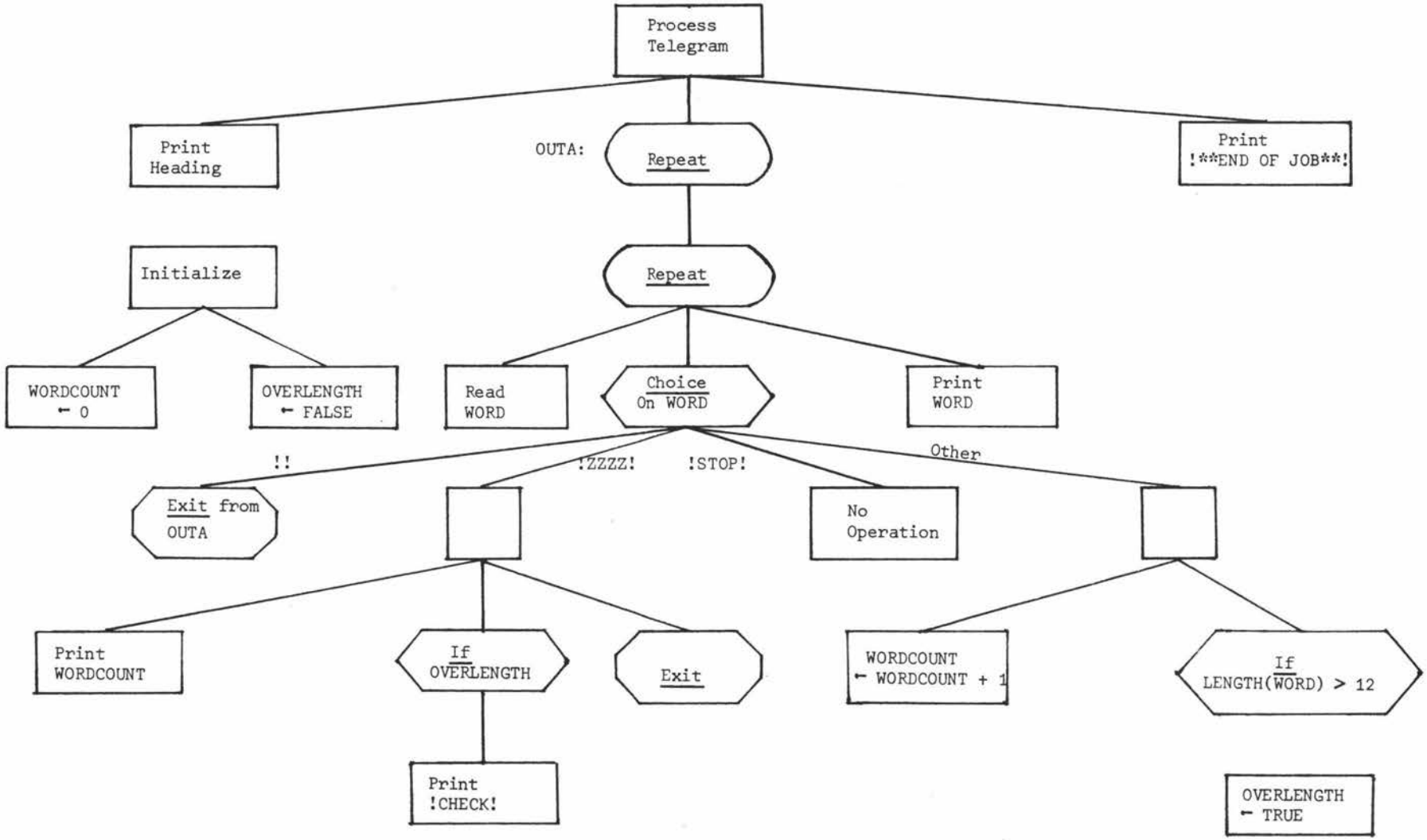


Fig 2.1 - Structure Diagram For Telegram Processing Example

The program

```
≠NAME   BROWN,CHARLES K.
*
*PROGRAM TO PROCESS STREAM OF TELEGRAMS
*
DO
     RESERVE WORDCOUNT,OVERLENGTH,WORD
     PRINT NEWPAGE,TAB(30),!TELEGRAM ANALYSIS!,
     NEWLINE,TAB(30),!**************************!,NEWLINE(3)
OUTA:DO REPEAT
          SET WORDCOUNT TO 0
          SET OVERLENGTH TO FALSE
          DO REPEAT
               READ WORD
               DO CHOICE OF
                    IF WORD .EQ. !! THEN EXIT FROM OUTA
                    IF WORD .EQ. !ZZZZ!
                    THEN DO
                              PRINT NEWLINE,!WORDCOUNT = !,
                                    WORDCOUNT(PIC=***)
                              IF OVERLENGTH
                              THEN PRINT SPACE(5),!**CHECK**!
                              EXIT
                         END
                    IF WORD .NE. !STOP!
                    THEN DO
                              SET WORDCOUNT TO WORDCOUNT+1
                              IF LENGTH(WORD) .GT. 12
                              THEN SET OVERLENGTH TO TRUE
                         END
               END
               PRINT WORD, SPACE
          END
     END OUTA
     PRINT NEWLINE(3),!**END OF JOB**!
END
```

CHAPTER 3

THE METHOD OF IMPLEMENTING MUSSEL

3.1 Introduction

Having designed the language MUSSEL we were then presented with the problem of how to implement it on a computer. At Massey University two computers are available, namely the IBM 1130 Computer and the IBM 1620, Model 2 Computer. The 1130, although a more modern machine than the 1620, has a limited character set available on the line-printer and also its character handling capabilities are not as good as on the 1620, so it was decided to implement MUSSEL on the 1620.

There were two basic objectives to be considered in the implementation, these are as follows:

(i) Implementation to be as machine independent as possible.

(ii) The implementation must have the ability to incorporate satisfactory error detection and error diagnostics.

With regard to the first of these objectives - the implementation on the 1620 is a short term measure only, and the ideal of machine independence has its limitations. For example the character set available is limited so that on another computer one would take advantage of extra characters being available. Also the 1620 is a variable word length decimal computer, which differs from the majority of computers in existence today. Thus, in detail, many features of the 1620 have influenced the implementation of MUSSEL.

To attain the second objective, MUSSEL has been implemented as an interpretive system i.e. the source language is translated into an internal form,called the Intermediate Language, during the Compilation Phase, and then this internal form of the program is interpreted by the Interpretive Phase.

The reasons influencing this type of implementation are the needs in an educational environment to have a system that provides

(i) fast compilation time

(ii) good diagnostic facilities,

both at compilation time and execution time.

A interpretive system can achieve both of these requirements. An interpreter of course, is much slower in execution than an equivalent language program, thus, it should not be used in general in a production environment, but for student programs the execution time is a trivial matter, so this is not a great drawback. One of the main advantages of an interpreter over a compiler is the ease of providing debugging facilities to the programmer. For example, it is fairly simple to implement an interpretation-time subroutine which, given the address X of a symbol table entry, prints the associated source program identifier together with its current value. Also whenever an interpretation - time error occurs it would be possible to print each symbol table entry with its corresponding value when the error occurred. This dump of the symbol table is of course, a symbolic one at the source program level. This type of dump is possible, but, more difficult to implement in a conventional compiler. By the insertion of a line number operator within the internal representation of the source language it is possible to give the source language line where an error has occurred at execution time,and with a little extra effort the source language statement itself could be listed. Trace facilities are also easy to implement in an interpretive system and this is another useful debugging aid to programmers.

3.2 The Compilation Phase This phase is like the first part of a multi-pass compiler.

The internal form of the source language, which is produced during this phase is called the Intermediate Language and is essentially a 'Reversed Polish' form of the original source program. Two listings of the program are given at this time - the first corresponds exactly to the submitted program, except the line number has been prefixed to each statement in the source program. The second listing is an amended form if necessary with correct indentation of statements and error messages. The recogniser itself is syntax-directed, top-down and uses a syntax graph with links to the associated language productions. MUSSEL being designed as a teaching language must have good error diagnostics. Because of the time factor it has not been possible at this stage to implement other than simple error diagnostics, but later the parsing algorithm will have to be altered to cope with these added facilities. Using a top-down method for parsing it is difficult to keep track of exactly where an error has occurred - this is more precisely pin-pointed using a bottom-up method, so it is conceivable that this may have to be introduced in order to effect good error detection later.

3.3 The Intermediate Language

The Intermediate Language (see Appendix C) produced by the compiler is similar to that used by Randell and Russell [1] being essentially 'Reverse Polish' notation consisting of a set of operations which the Interpreter obeys. The Intermediate Language of MUSSEL is, however much simpler than that adopted by Randell and Russell since MUSSEL is of a simpler structure than Algol. Because MUSSEL has no specification for type it is not necessary to introduce type operations or conversion operations into the Intermediate Language.

3.4 The Interpretive Phase

The Interpreter phase in MUSSEL ,which has been fully described in P.B. Gibbon's M.Sc. Thesis, entitled: "The Design and Implementation of a Structured Programming Language with Few Arbitrary Restrictions - the Interpretive Phase" ,uses an execution time stack for evaluation of expressions and dynamic storage allocation. It will be possible in this phase to introduce good error diagnostics, because access to the names of variables and procedures ,and indeed to the source language itself ,has been provided for in the compilation phase and these are all available on disk back-up. The line number of the original source code is incorporated into the Intermediate Language. By this means, the Interpretive phase will be able to keep a watch on all execution-time errors and produce good error messages accordingly.

CHAPTER 4

THE DESIGN OF THE COMPILATION PHASE

4.1 Introduction

The task of the Compilation phase is to convert the MUSSEL source text into Intermediate Language operations, which are then obeyed interpretively by the Interpretive phase.

The Intermediate Language generated in this phase, makes no attempt to be machine dependent - it is merely a string of operators and operand references, in an augmented "Reversed Polish" notation, which is sufficient for the Interpretive phase to efficiently analyse and completely process (see Appendix C).

The Compilation phase consists of three distinct modules

(i) The Main Control and Input Module.
(ii) The Parsing and Intermediate Language Generation Module
(iii) The Interface Module.

4.2 The Main Control and Input Module

Since MUSSEL is to be used primarily as a teaching language, then any implementation of the language must be designed to run in a Batch Mode i.e. every usual run of the System will consist of a stack of programs to be run one after the other.
The Main Control and Input Module has three main functions to perform.

(i) To initialize System for the start of a new Batch Run
(ii) Control flow of jobs through the System
(iii) Provide Input to the Parsing and Intermediate Language Generation module.

At the commencement of a new batch run, the System is initialized and the Controller asks for and accepts the date. This date is stored for the duration of the run, and it appears at the top of each page of each student job.

4.2.2. Control Cards: To Control the flow of jobs through the system it is necessary for each job to have control cards, before it, these consist of

(i) name card This is punched with ≠NAME in the first 5 columns of the card, followed by the students name, which may be punched anywhere in the remaining 75 columns of the card. This card is a signal to the Control module that a new job is on its way. On reading a name card the Controller sets up the system for a new job and extracts the students name for appending to the program listings later and then goes back to read another card.

(ii) code card This is punched with $CODE in the first 5 columns of the card. This control card is included optionally, with the program if the student requires a listing of the Intermediate Language, after it has been generated in the Parsing and Intermediate Language Generation module. The Controller on detecting this card in the job stream, sets a switch, which will be tested later at the commencment of the Interface module.

(iii) symbol table card This is punched with $XREF in the first 5 columns. This control card is included as an option to have the Symbol Table listed at the conclusion of the Interpretive phase, or during it if an error should occur. Likewise, as with the code card, the controller on detecting this card in the job stream, sets a switch, which will be tested in the Interpretive phase.

(iv) data card This is punched with ≠DATA in the first 5 columns of the card. This card appears at the end of a program and before the data if any. This card signifies the end of a program and the Control module on detecting it in the job stream, writes end of program record to disk and Sets up heading for the Parse and Intermediate Language Generation Module to use. The symbol table is also initialized at this stage, before linking to the Parse and Intermediate Language Generation module.

(v) end of batch card This card is punched with four record marks (≠) in the first four columns of the card. This card signifies the end of a batch run and the Control module on detecting it types a message on the console typewriter, indicating the end of the batch run, and also at this time user statistics could be listed.

4.2.3 Input Section: Besides controlling the flow of jobs through the system, this is also responsible for providing the Input to the Parsing and Intermediate Language Generation Module.

The source program is read from cards, the superfluous blanks are trimmed off the end and an internal end of line character inserted where applicable. The punching codes for the characters ≠, ! and : are converted to their internal line-printer representation and a line number is associated with each source line, provided it is not a continuation of a previous line. The amended source text is then listed exactly as punched, and written off to disk for input to the Parsing and Intermediate Language Generation Module. The general flow of this Main Control and Input Module is included in Appendix E.

This module could have been extended to include a lexical analysis of the source text i.e. it could have been designed to scan the characters of the source program from left to right and build the actual symbols of the program i.e. identifiers, reserved words, numbers to their internal representation, character symbols e.g. **, + etc. The reason this method was not adopted was, because it was necessary to have a copy of the original source text available, both at the actual syntactic analysis time, and later at the interpretive stage for meaningful error messages to be produced. It was therefore decided to do the lexical analysis in the Parsing module via a subroutine called SCANS, this subroutine is called by the syntax analyser, whenever it needs a new symbol.

Omitting the lexical analysis at this stage, has made the Control and Input module a very simple phase, with all the hard work left to the Parsing and Intermediate Language Generation Module.

4.3 The Parsing and Intermediate Language Generation

This module performs all the real hard work of the Compilation phase. Both the lexical analysis and syntactic analysis have to be performed at this stage, together with the semantic actions when a construct in the Source Language is recognised. The semantic actions consist of checking for semantic correctness, storing information in the Symbol table or output of Intermediate Language operations and parameters into the Intermediate Language stream, for example when a RESERVE instruction is recognised a semantic routine will check the identifiers in the RESERVE 'list' to make sure they have not been used before and if not will add them to the Symbol Table with their attributes (if any).

The general layout of core during this phase is depicted in Fig 4.1.

4.3.1 The Lexical Analysis The lexical analysis is performed by routine called SCANS. This routine is called by the Syntax Analyser everytime it needs a new Symbol. When it is called SCANS recognises the next source language Symbol and passes it to the Syntax Analyser. In this way, it is not necessary for the whole of the internal form of the Source program to be in core at the same time, as would have been the case if the lexical analysis had been performed in the Main Control and Input Module. The Symbols in MUSSEL fall into one of the following classes:

(i) identifiers
(ii) reserved words - DØ,END,CHØICE,CASE etc.
(iii) numbers - no distinction is made between real numbers or integer numbers.
(iv) single character delimiters (+,-,(,),/,:,!,*,.,)
(v) double character delimiters **
(vi) relational, logical and string operators - .LT.,.AND.,.CAT.etc.

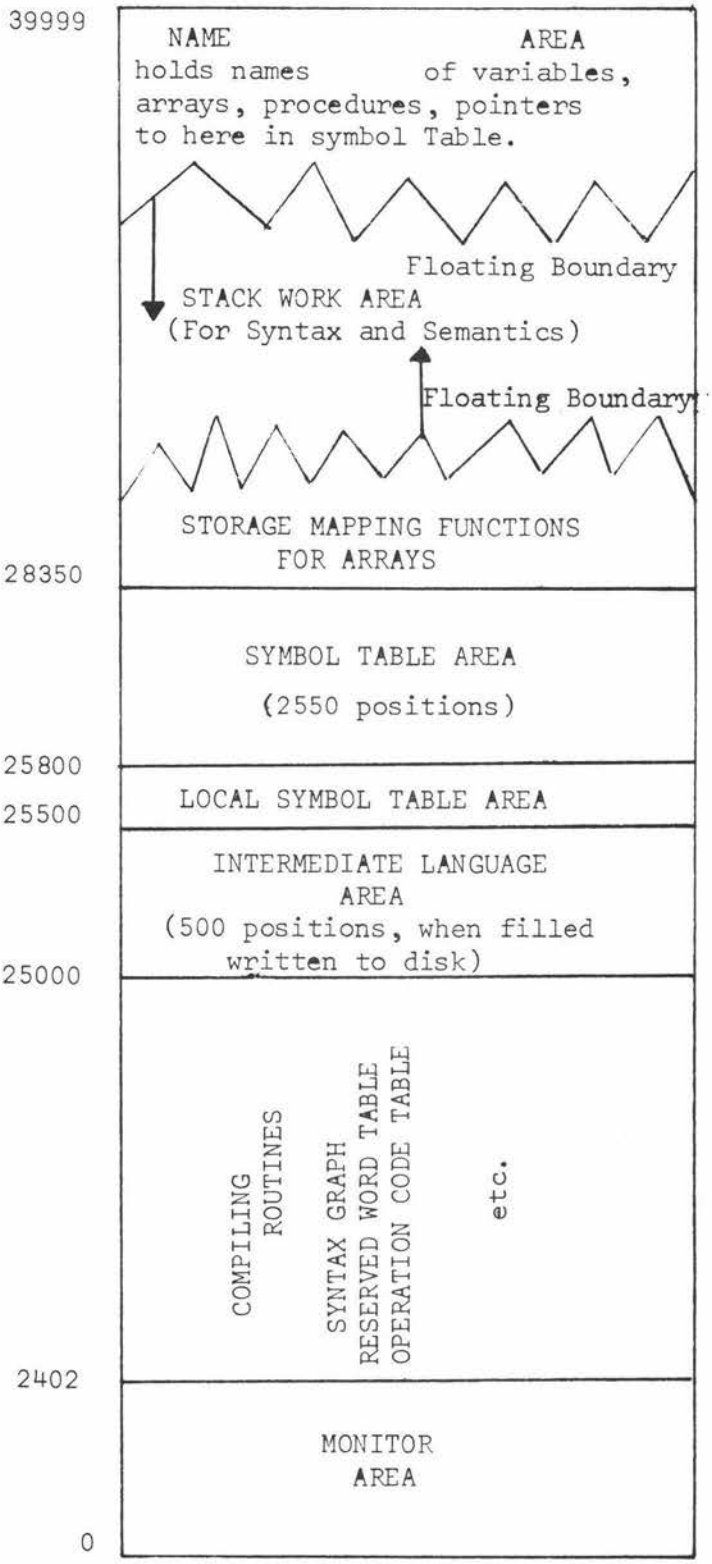


Fig 4.1 Storage Layout During Parsing and Intermediate Language Generation

The SCANS routine has encoded within it many routines for resolving the various source symbols. These routines are:

(i) CDC Routine This routine gets the next character,CHAR from a source language record and performs the following functions.

(a) Ignore blank characters - this is handled by a routine to get next non blank character,CCCH,which uses the CDC routine.

(b) Obtain the source character and make it available to the routine requesting it within SCANS.

(c) Prints a source language line on picking up the internal end of line character (≠)

(d) Keeps track of where in the source text the compiler is operating by means of CPTR.

(e) On detecting the end of a source record, prints it and automatically fetches the next record and updates CPTR.

(ii) GNS Routine This routine picks up the next symbol in the source text, which, may be a reserved word,number,operator,identifier etc. Each type of symbol in the language has a special routine to process it. Thus, GNS by means of the CDC routine picks up enough characters to resolve which type of symbol it is and then turns control over to the special routine for that symbol to be processed. These special routines are:

(a) ALF Routine GNS having detected the first character of a symbol as being a letter, turns control over to the ALF routine, which, initializes the name holding area, NACC to blanks and then by means of the CDC routine builds up the name character by character, until detecting an end of name character (i.e. a character other than a letter or a digit).

As mentioned previously MUSSEL imposes no restrictions on the length of any name - for this reason the ALF routine has to keep a record of the length of the name as it is accumulated. Provision is made in NACC to hold a name up to 25 characters in length, if this is exceeded then the

first 25 characters are moved into the name holding area at the top of core, NACC is set back to blanks and the rest of the name accumulated.

(b) NUM Routine This routine picks up a number and converts it to its internal representation. MUSSEL makes no distinction in representing integer numbers and real numbers. Integer numbers are distinguished from the reals by having a zero exponent. Numbers are represented internally by a seven digit mantissa and a 2 digit exponent, with the decimal point being implied to the right of the mantissa as opposed to the usual representation, where the decimal point is implied to the left of the mantissa. Thus the integer 5 is represented internally as $\bar{0}00000\bar{5}\bar{0}0$ and 27.6998 as $\bar{0}2769980\bar{4}$
The function of the NUM routine is then to

(a) Initialize the number holding area, NUMB to zero

(b) accumulate the number in NUMB digit by digit.

(c) deal with decimal point occurring within number.

(d) deal with exponent (E) occurring within number

(e) detect illegally large numbers.

(f) distinguish between real and integer numbers.

(g) detect end of number.

(c) GOP Routine The function of this routine is to resolve operators, such as +,-,*,/, also the multiple operator ** for exponentiation. The relational, logical and string operators are all depicted by delimiting them on either side by a decimal point viz .NE.,.NOT.,.CAT.etc.
These operators are also resolved using the GOP routine.

(c)cont'd Each operators hierarchy number is obtained via table look-up, and this together with a pointer to the operating holding area is returned for generation later into the Intermediate Language stream.

The SCANS routine, thus furnishes to the syntax analyser an internal representation of each symbol in the source language, together with a pointer to the actual symbol itself. This is more efficient than returning the variable length strings of actual symbols to the syntax analyser.

4.3.2 Indentation A programmer writing in MUSSEL must indent his source text correctly according to precise indentation rules. The reasons for making these conventions in MUSSEL quite explicit and precise is that indentation can be closely linked to the flow of control and therefore becomes a guide to

(i) the program writer as to whether he or she has handled control sequencing correctly and as intended.

(ii) to the reader of a program as to how the sequence control may be understood. Indentation in MUSSEL is readily accomplished due to the absence of goto's in the language and the absence of labels except on the head of instruction groups with the special mechanism available via the EXIT FRØM instruction of terminating execution of such a group. If this were not so the subordination of text may not be as clear. We hope that the rewards for enforcing these rules are

(a) that the program will be more likely to be correct because conditions must be considered carefully.

(b) The structure of the source text will reflect the flow of control at execution time, thus making the program more readable. The general flow of the program will be indicated on the least indented lines and detailed flow is shown by the lines in between. In fact the structure of the text is a mirror of the structure diagram, which, was drawn in developing the algorithm initially.

Also having set down precise rules for indentation, has enabled the indenting to be checked and performed automatically where necessary during the Parsing and Intermediate Language generation phase. Thus the listing of the source text in this phase is correctly structured regardless of the way it was actually punched onto cards.

The goals for introducing an indentation convention were

(i) Changes of level of indentation should coincide with important and recognisable constructions in the algorithm.

(ii) Avoid needless indentation as this only results in crowding of the algorithm to the right.

(iii) The rules should be simple and easy to carry out.

In MUSSEL we have achieved these goals by setting down the following indentation rules.

(a) At the occurrence of each DØ which signifies the start of an instruction group, indent by five spaces to the right.

(b) At the occurrence of each END, which signifies the finish of an instruction group, indent five spaces to the left.

As was stated previously, in MUSSEL the only instruction that may be labelled are the DØ's, being the head of an instruction group. This means that if the first instruction in a MUSSEL program is labelled then the label must appear on the previous line, likewise all labels that will not fit in the space available, viz.

```
PRØGRAMSTART:
DØ
    RESERVE X,Y,Z
    DEFINE A ØN C,D,E AS
PRØC:DØ
        RESERVE L,M,N
        ..............
     END PRØC
      ...............
END PRØGRAMSTART
```

In the conditional groups of instructions a DØ may be prefixed by THEN or ELSE and the indentation by 5 still holds.
For example.

```
DØ IF I.EQ.0
THEN DØ
      SET S TØ N+K
      SET P TØ P+S
     END
ELSE DØ
      SET S TØ N-K
      SET P TØ P-S
     END
END
```

4.3.3 Symbol Table

The Symbol Table during the compilation phase is organised as depicted in Fig 4.2. Each position in the table occupies seventeen digits made up as follows:

(i) Name Pointer This occupies four digits and represents a pointer, relative to a base address, to the actual name of the identifier which is stored in the name area at the top end of core storage - see Fig 4.1.

(ii) Overflo' This occupies three digits and represents a pointer (if any) chaining names together in the event of them hashing to the same position in the Symbol Table. Details of this procedure will be referred to later under Symbol Table Look Up.

(iii) Type Occupies one digit and holds the type of the identifier stored in that position of the table. As has been mentioned previously, a feature of MUSSEL is the absence of explicit type declarations, each variables type is set implicitly during the Interpretive Phase. This allows for dynamic type change throughout a program. During compilation the only types that are known and set are for procedure names (type code = 5), labels (type code = -1) and array identifiers

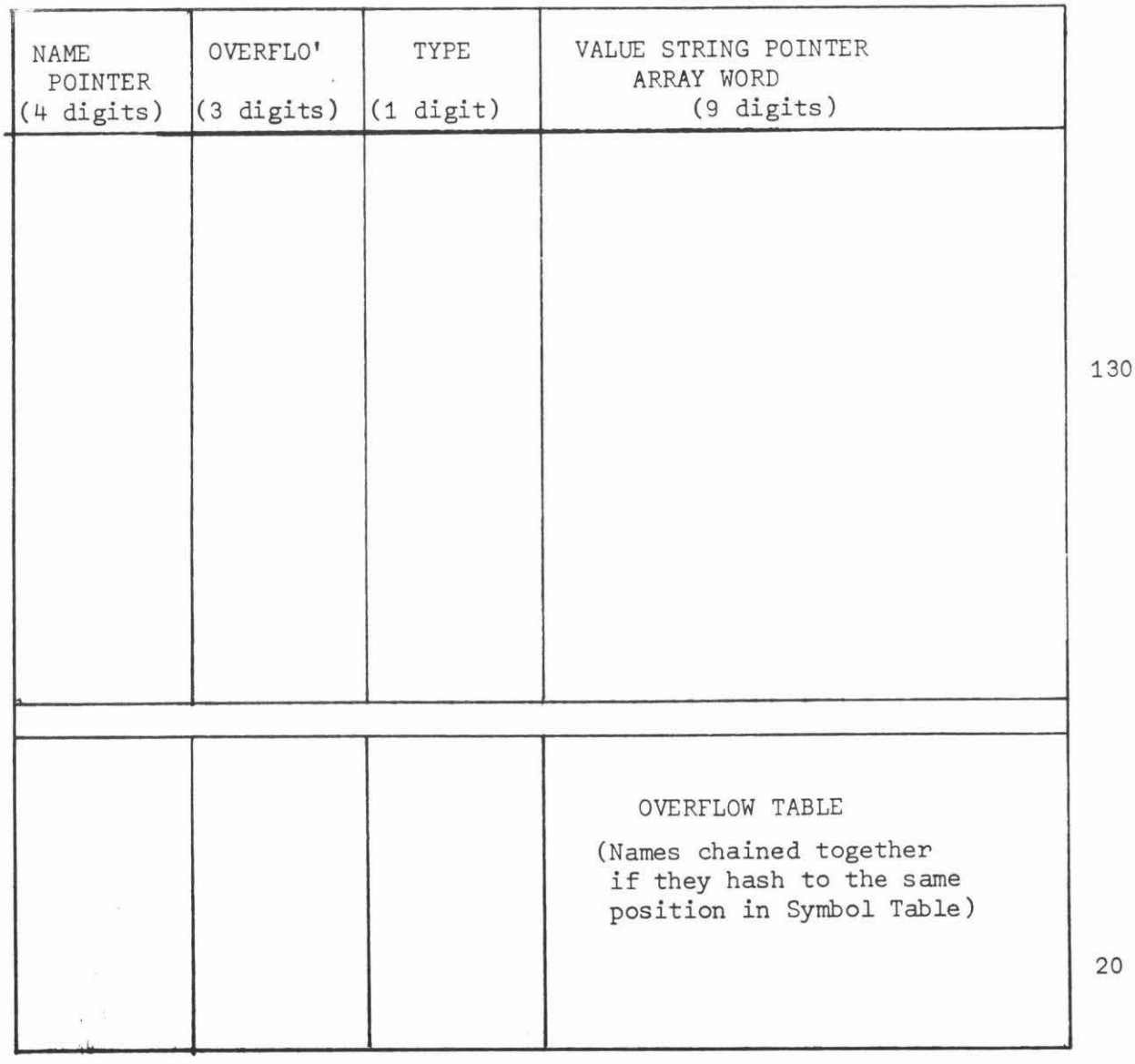


Fig 4.2 Symbol Table Layout During Compilation Phase

reserved at the head of the main program (type code = 4). All other identifiers have a type code of zero.

(iv) Value, String Pointer, Array Word Occupies nine digits and is the area used to hold the value of a variable during the interpretive phase. During the compilation phase this area is zero except for the positions in the table occupied by main program array identifiers, procedure names and labels. For main array identifiers during compilation the array word is stored in this area and is of the following form:

$$D\bar{E}EEE\bar{S}SSS$$

where D signifies the dimenstion of the array, which is either 0 for one dimensional arrays and 1 for two dimensional arrays. $\bar{E}EEE$ signifies relative address of the first array element. This is added to the Symbol Table base address to give an absolute address for the first array element. $\bar{S}SSS$ signifies the relative address of the storage mapping function for the array. These storage mapping functions are stored immediately following the Symbol Table as indicated in Fig 4.1.

Because arrays can only be 1 or 2 dimensional in MUSSEL, the storage management is greatly simplified, and the storage mapping functions may be one of the following two forms:

(a) one dimensional storage mapping function

$$\bar{L}LLL\bar{U}UUU$$

lower subscript bound — upper subscript bound.

(b) two dimensional storage mapping function

$$\bar{L}_1L_1L_1L_1\bar{U}_1U_1U_1U_1\bar{L}_2L_2L_2L_2\bar{U}_2U_2U_2U_2X\bar{C}CC$$

1st lower bound(L_1) — 1st upper bound(U_1) — 2nd lower bound(L_2) — 2nd upper bound(U_2) — U_2-L_{2+1}

For Procedure names this 9 digit area contains the procedure address word, which contains a pointer to the start of the Intermediate Language corresponding to the procedure body, and a pointer to the first local name held in name area.

For labels this 9 digit area contains a pointer to the Intermediate Language stream where the exit from that loop is contained.

There are one hundred and fifty positions allocated for the Symbol Table in the present implementation, and of these one hundred and thirty of these positions represent the Symbol Table proper and the remaining twenty positions represent the Overflow Portion of the table.

4.3.4 Symbol Table Look Up

Of vital importance to the efficiency of the compiler is the method adapted for accessing the Symbol Table, as this table must be looked up on each reference to any identifier in the program. In MUSSEL the technique used for Symbol Table Look Up is called the "hashing" or "key" transformation method. Here the search key, which for Symbol table access is the name of an identifier is converted or hashed into a direct address into the table. This address is then used to locate or place the proper data item. There are a great many variations and many possible schemes, which may be adopted using this method. The one chosen for use in MUSSEL is the division-remainder method, which operates as follows:

First choose a number close to the number of table positions needed, in MUSSEL's case the number is 130. Next use this number as a divisor to extract a quotient and a remainder from the original key (name). The remainder is the transformed key i.e. the transformed key becomes the original key modulo the chosen number of table positions.

i.e. $h(N) = N \bmod M$

where N is the original, M the table size and h(N) the hashed function.

Ideally, if the transformation process is unique, no two names would produce the same transformed key, and no unused positions would occur in the table.

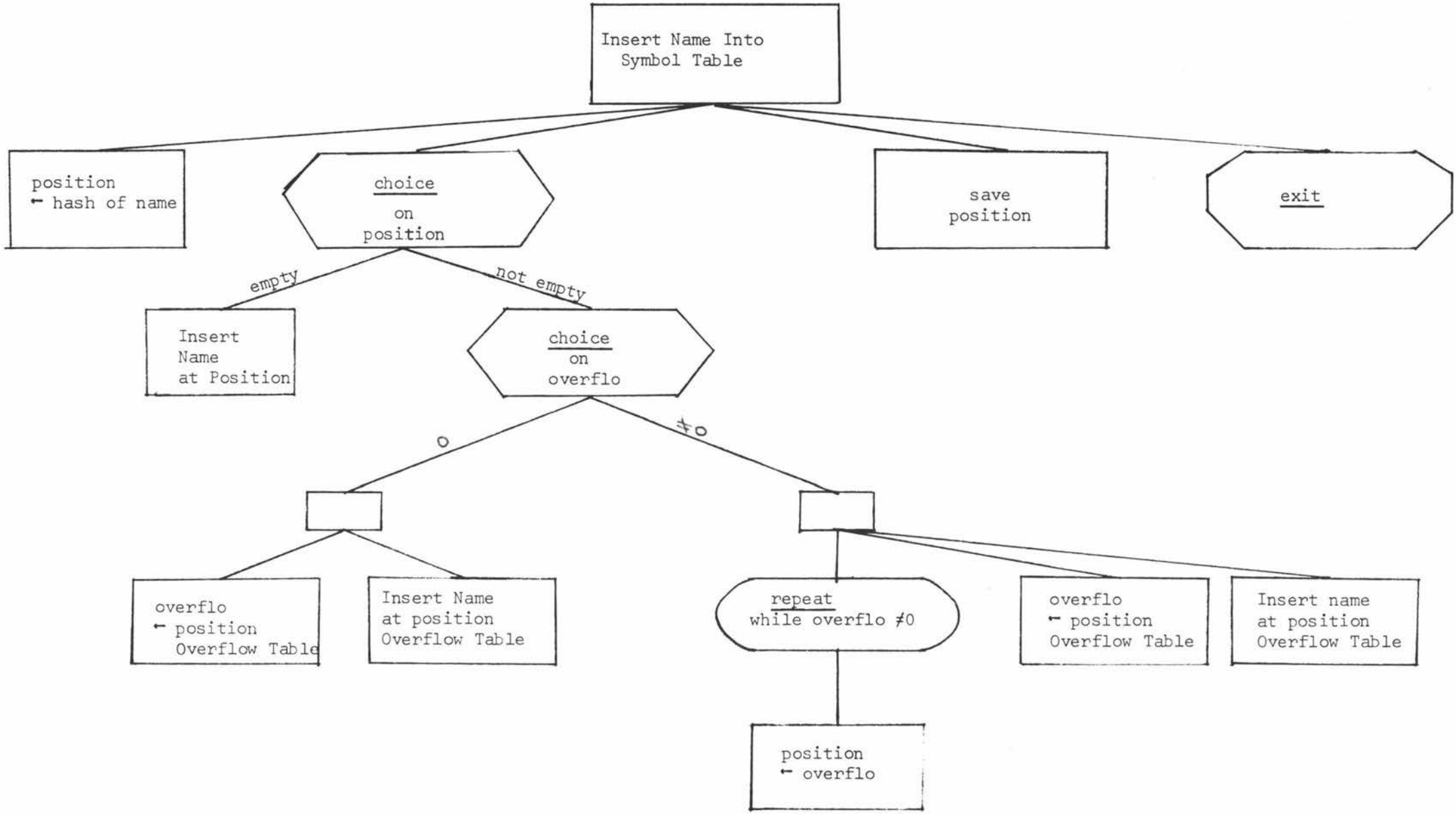


Fig 4.3 Inserting Name into Symbol Table.

While it is possible to devise schemes to minimize these "hash clashes" its possibility can never be eliminated. Because of the probability that duplicate keys will be generated by the transform method, the original key or pointer to it, as in MUSSEL's Symbol Table, must be stored in the table. Also the resolution of duplicate keys or "hash collisions" has to be decided on.

In MUSSEL "hash collisions" are handled simply by using an Overflow Table, which operates as follows.

If when placing a name in the table the transformed name finds an empty slot in the Symbol Table itself then the name together with its' attributes (if any) is placed there. If the slot is already occupied by a previously transformed name, then the overflow portion of this slot may be constructed to be a position in the Overflow Table. This "chains" the look up directly to the overflow table position containing the overflow key. Thus in essence the overflow table becomes a direct entry table, like the Symbol Table itself. Each position or slot in the overflow table has a overflow area, so the chaining process can be continued within the overflow table. The last one on the end of a chain has a zero in the overflow portion and this signifies the end of the chain.

This method results in a faster look up than a trial and error overflow table search, or a sequential search within the main symbol table.

Of course in adopting this method of approach it is necessary that enough space must be allocated to provide for all overflows, and therefore it is inefficient on storage space.

In many references it is stated that a prime number should be chosen for the table size when using the division - remainder method. In general, it would seem there is no reason to believe why a prime number divisor should produce fewer collisions than a non prime number. Obviously the choice of number as the divisor could lead to substantial bias in many cases, for example if number chosen is even then the hashed key will be even when the original key is even, and odd when original key is odd. Also choosing a number as divisor which is a power of the radix of the computer being used could produce many collisions.

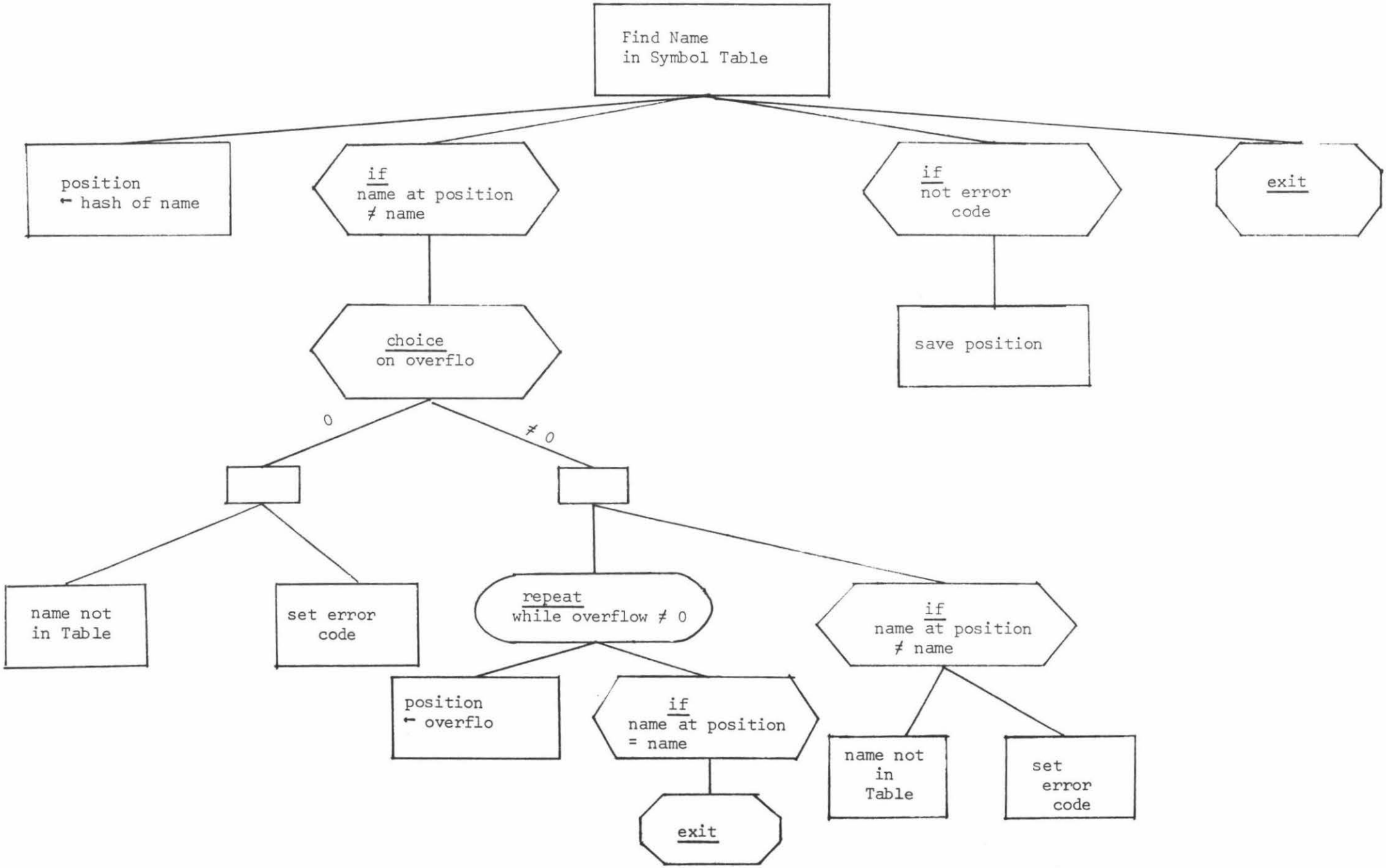


Fig 4.4 Searching for Name in Symbol Table.

In using this method it should be noted that the symbol table must be thought of as starting from zero and not one. In MUSSEL, since the interpretive phase has assumed that the symbol table starts from one it has been necessary to add one to the hashed key value. On a decimal computer like the 1620 many of the methods of "hashing" are not suitable. There were three other methods I considered as well as the division - remainder method, these were

(i) Splitting the original key into 2 or more parts and then adding and discarding digits.

(ii) Changing the base of the original key and extracting part of the transformed number

(iii) Reversal of digits of the original key and discarding digits.

All these 3 methods seemed more applicable to a table size larger than could be allocated in this implementation and would not have been very much faster.

It should be noted that using the division - remainder method identifiers with names, such as ABC, BAC, CBA all hash to the same position in the symbol table.

Example Illustrating Overflow Handling

SYMBOL TABLE (Has 5 positions)

Position	Original Key	Hashed Key	Overflo'
0	135(XY)	0	1
1	126(ABC)	1	3
2			
3	82(AB)	3	0
4			

OVERFLOW TABLE (4 positions)

Position	Original Key	Hashed Key	Overflo'
1	55(N)	0	2
2	85(AD)	0	0
3	41(A)	1	0
4			

4.3.5 Reserved Word Table

In many compilers the reserved words are "hashed" directly into the symbol table for ease of access. In the case of MUSSEL this method was not considered practicable, due to the fact that MUSSEL is an interpretive system and the symbol table is thus carried over into the Interpretive Phase. Adopting this method would have meant that the symbol table would have had to be of a much larger size than is at present allocated, introducing a large overhead into the System.

MUSSEL uses instead an alphabetically ordered reserved word table and a generalised binary search routine for accessing it. This routine obtains a pointer for a transfer vector, which then branches to the appropriate routine for a particular reserved word.

This method has been adopted also for the library function table and the special operator table.

4.3.6 Syntax of MUSSEL

In designing a new programming language due consideration must be given to the syntax structure of the language, as this will have a profound influence on the parsing method used to analyse it.

The guiding idea in the syntax of MUSSEL was simplicity, due to the recognition that we required a structure not too difficult to master by novice programmers, and thus because of its simplicity not too difficult to process on a computer.

In MUSSEL we required that the syntax be such that it would be possible to meaningfully diagnose syntactic errors and be able to continue processing subsequent source text with a reasonable degree of certainty of arriving at a correct diagnosis - this capability is essential where a language is to be used by first time programmers.

All instructions in MUSSEL begin with a unique key word. This property facilitates the processing by computers and aids the understanding of programs by human readers.

The syntax of MUSSEL in Backus Normal Form (BNF) is given in Appendix A.

The compilation of MUSSEL source language to Intermediate Language, that is interpreted, is achieved by a syntax - directed method. That is rather than having the syntactic structure of the language reflected in the actual encoding of the compiler, the syntactic structure is encoded as a list structure called a syntax graph. Syntactic analysis is then achieved by having a routine (syntax driver) to step through this syntax graph.

4.3.7 Method of Parsing MUSSEL

Having designed a new programming language and then implemented it on a computer, it is inevitable that modifications to the language will result, when it comes to be used. This fact had to be borne in mind when deciding how to parse the MUSSEL source language.

Two methods of approach were considered.

(i) Recursive Descent Method.

In this method the parser or syntax analyser has one recursive routine for each production in the language. The routine is told where in the program to start looking for a phrase for a particular nonterminal, N and hence it is goal oriented. The routine finds its phrase by comparing the program at the point indicated with the right parts of rules for N and calls on other routines to recognise subgoals where necessary.
The Semantic action to be taken on recognition of a production is built into the recursive routine. Hence this method is very flexible to change. This method is ideal if one is writing the compiler in a high level language such as Algol, but would be difficult to implement using assembler language, as it would be necessary to set up the mechanism for recursive calls.

(ii) Top-Down Syntax-Directed Method

In this method the syntax of the language is encoded directly and a syntax control routine or syntax driver is used to step through the syntax starting with the initial goal of the syntax and attempting to generate a string of terminal symbols, which matches the source language statement being translated.

One of the problems with this method, especially on small computers like the 1620, is representing the syntax in the machine. One method used by Cheatham and Sattley [28] was to represent the syntax by a pair of tables and this is the usual method of approach. On small computers syntax tables are not particulalry suitable, the reason being that they occupy a large amount of core storage with a great many entries in the table being empty. Another method of representing the syntax, due to Cohen and Gotlieb [14] is in a list structure form known as a syntax graph, in which pointers are used to link the various components and alternatives of a production. This method of representation is economical on core storage and too, the parsing algorithms are less complicated. This has been the method adopted in MUSSEL and results in a flexible system, which may be easily altered at any stage.

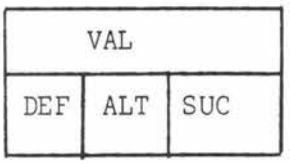
4.3.8 Construction of the Syntax Graph

To construct the syntax graph for MUSSEL all productions starting with the same nonterminal are combined to form a tree for example consider the nonterminal, valexp of MUSSEL is written in BNF form as

<valexp>::= <term>|<adding op> <term>|<valexp> <adding op> <term>

where | separates the different alternatives.

The set of all such trees forms a disjoint set of subgraphs of the syntax graph. These trees, together with some connecting links constitute the syntax graph. Each node of the syntax graph may be represented pictorially by a quadruplet



(a) VALue or name, which is one of the elements of the vocabulary set of the MUSSEL grammar (either terminal or nonterminal)

(b) DEFinition Pointer, which is the interconnecting link of the trees. Terminal nodes have no DEF link and are depicted by the special symbol, $\Downarrow$.

(c) ALTernative Pointer The different alternative productions in each tree, are linked by means of the ALT link of the first element of each production in the tree. The last alternative is indicated by φ in ALT link.

(d) SUCcessor Pointer This links the elements of each production from left to right. The last component of each production has the special end of production symbol, σ in its SUC link. A flag at the side bearing the name of the nonterminal is attached to the root of the tree.

Thus the elements of a production are linked through their SUC links with the symbol σ signifying the end of each production. A production is referenced by its first element, so that the different productions are connected through their ALT link of the first element of each. The last alternative in a list of alternatives for the nonterminal defined by the tree has φ for its ALT link. Fig 4.5 illustrates the tree for valexp

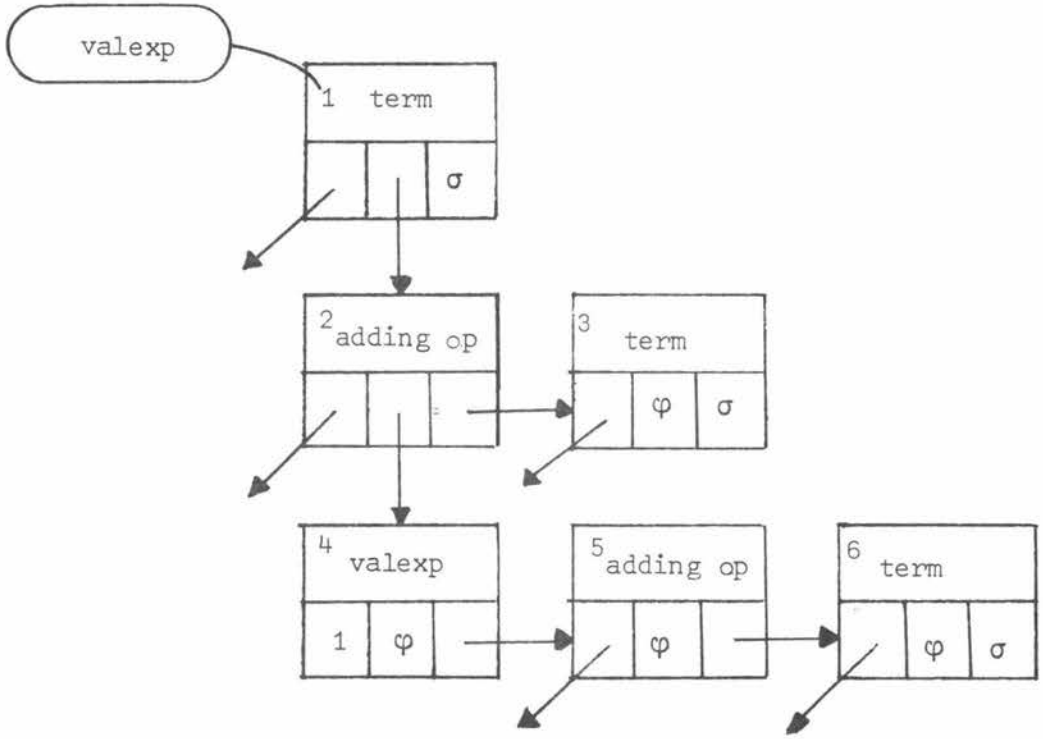


Fig 4.5 The tree for the nonterminal Valexp of MUSSEL.

Each node within the syntax graph may be considered as a decision point, whereupon if the symbol currently being analysed matches the node one route in the graph is chosen. If however, the symbol does not match the node, an alternate route through the graph is taken, or if there is no alternate route an error condition is recognised.

In constructing the syntax graph for MUSSEL it was necessary to firstly produce a formal syntax of the language in BNF form - see Appendix A. Using this formal syntax, then build a tree for each nonterminal and connect each tree by means of the DEF links. The construction of the graph is then completed by creating the root node (R) of the graph. This is the initial syntax symbol, which in MUSSEL's case is PROGRAM and it is linked to the tree defining it by its DEF link while its ALT link is φ and its SUC link is σ. This syntax graph contains all the information about the syntax of the language and is in a nonreduced form, which may be simplified, whereby reducing the number of nodes required to represent the syntax. The reduced form of the syntax graph for MUSSEL is given in Appendix B. The reductions that may be applied to the nonreduced form of the syntax graph are as follows and are applied in the order given.

(i) Factoring Reduction This takes advantage of identical components within a tree and may be factored out.

(ii) Elimination Reduction This reduction eliminates those nonterminals, whose trees are made up of terminal components only, for example : >::=+/-

(iii) Recursion Reduction This reduction recognises recursive components and simplifies the graph accordingly and presents infinite looping.

(iv) Redundacy Reduction This reduction eliminates nodes whose value and links are identical to the value and links of other nodes.

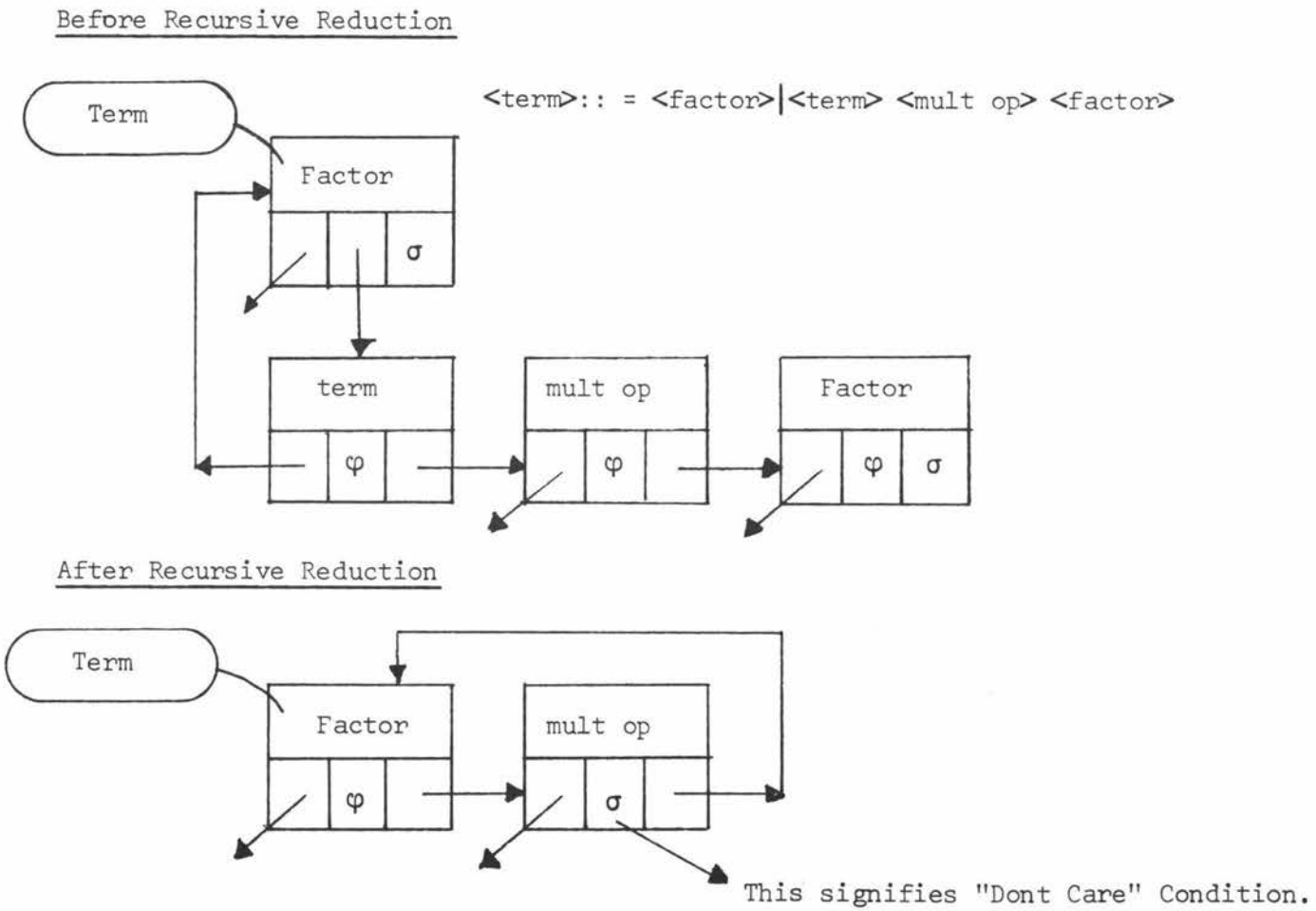


Fig 4.6 Example Illustrating Recursive Reduction.

Dont Care Conditions These are signified by a σ in the ALT link in the last link of a list of alternatives. It means that a match of this node is not necessary for the recognition of the nonterminal defined by this production. It should be noted that by applying reductions a certain loss of information occurs, which is initially included in the nonreduced graph and for semantic reasons it is not always advisable to reduce the graph too far, as we begin to lose sight of the original syntax.

4.3.9 Representation of the Syntax Graph in the Computer

To represent the syntax graph internally in the machine, each node has associated with it the three pointers DEF,ALT and SUC. For terminal nodes the DEF link is set to zero, and also associated with terminal nodes is another pointer which links to the encoding of the terminal, and this in turn has a pointer to a routine to handle the semantic action to be taken upon recognition of this syntactic class. The ALT link for the last alternative in a production is set to zero and for "dont care" condition to 99999. The SUC link for the last component of each production has the value 99999. Nodes requiring some semantic action upon recognition, but are not terminal nodes,have a flag set to indicate another pointer to the node and this points to the routine which performs the necessary action.

4.3.10 The Syntax Control Routine

The syntax control routine, SCONER used in parsing the MUSSEL source code is a modified version of the Cheatham and Sattley algorithm [28] and is based on the syntax graph, rather than syntax tables. The routine cannot backtrack, which is a desirable feature, since backtracking makes parsing very slow. In recent years work done on parsing methods has focused attention on the methods which reduce or eliminate the need to backtrack.

A structure diagram for the routine is given in Appendix E.

In essence the syntax control routine calls on itself until a terminal node in the syntax graph is detected (DEF = 0). As it steps through the graph it places on the syntax stack the node pointer, G and a flag switch, F, which indicates whether the goal for this node is still being sought or not. When a terminal node is detected (DEF link = 0), it checks the current symbol held in CHAR (reserved word, operator, number, variable etc) with the value for the terminal, if a match occurs stacks node and branches to the semantic routine for this terminal. These semantic routines may cause the setting of certain switches for later use, output of code into the Intermediate Language stream or stacking of information into a semantic stack for fixing up later, also error conditions will be reported back. If a mismatch occurs at a terminal node, then G is reset to alternative for G,and cycle repeated. If G has no alternative (ALT = 0), then a previous G is popped off the stack and G is reset to the alternative for this node.

4.3.11 Semantic Routines

When the syntax control routine recognises a source language construct it branches to a semantic routine for that construct,which takes it, checks it for correctness and stores necessary information about it, either into the symbol table, intermediate language stream or it may result in stacking information into the semantic stack for fixing up later, also it may set certain switches for later use. For example, when a RESERVE is recognised the semantic routine sets a switch to indicate that it will be an insert into the symbol table and not a look up. It then checks the identifiers that are in the RESERVE 'list' to make sure they haven't already been placed in the symbol table and if not will add them to the symbol table.

As another example, at the recognition of a procedure definition, the semantic routine for this construct, outputs an unconditional jump instruction (UJ) to the intermediate language stream and stacks its position for later insertion of where it is to jump to at the conclusion of the procedure code. The name of the procedure is

entered into the symbol table, along with its type code and the address in the intermediate language where the procedure code starts. Parameters to the procedure (if any) are entered into the local symbol table, with their position relative to the return address word (RAW) in the stack at execution time. A procedure entry (PE) is output to the intermediate language with the number of parameters, the number of local variables is inserted later after the analysis of the RESERVE 'list' instruction within the procedure (if any).

4.3.12 Error Diagnostics

As indicated in the structure diagram of the Syntax control routine, see Appendix E an error condition can be returned to the system. This indicates a syntax error, such as punctuation errors, misspellings, illegal constructions within a source statement. Using the top-down method for syntactic analysis it is difficult to say when an error condition arises, what has actually caused the error. The only information available is that the position in the source statement, when the error arises, is known. As the syntax control routine steps through the syntax graph,structures within the statement are recognised and the SCANS routine advances to the next symbol. When an error condition arises the control routine has not advanced past the symbol that caused the error. When this condition occurs, an error is reported and a routine CYCLE is called to skip through the text until an end of statement occurs. Semantic errors i.e. those errors concerned with the incorrect use of identifiers and expressions are simply reported by a number message and nothing is done about error recovery. All tables and stacks are checked for overflows and a error message printed if this condition arises.

There is a large amount of work that could be done to improve the error diagnostics as mentioned later in section 5.5

4.4 Interface Module

The MUSSEL Interpretive System is a composite of the Compilation and Interpretive phases. The two phases interface or merge through the Symbol Table and Intermediate Language. It is in the Interface module that the merging takes place. In this phase, all or part of the Intermediate Language, which was generated during compilation is written off to disk. The names of all identifiers, which are held in the top end of core storage during compilation are written to disk, for possible future developments of the system via debugging and error diagnostics at execution time. The Symbol Table, together with the Storage Mapping Functions for arrays (if any) are shifted to take up a new position in core for the Interpretive Phase. The overflo position of the symbol table is dropped at this stage, thus reducing its size. Addresses for the start of the Intermediate Language, first array element (if any) and interpret-time stack base are set at this point, together with the disk address for the names. The Intermediate Language is brought back into core ready for the Interpreter to start execution.

This module has been extended during the implementation stages to include optionally a printout of the Intermediate Language in mnemonic form, and also a printout of the symbol table.

CHAPTER 5

FUTURE DEVELOPMENTS OF THE MUSSEL SYSTEM

The system that has been developed on the 1620 has only really laid the basic foundations to the system that was envisaged at the outset. MUSSEL has been designed with one purpose in mind of being a possible language to teach to first year computer science students, and as such, the prime features of any implementation of it should be

(a) good error diagnostics with meaningful error messages, both at compile and execution time.

(b) debugging facilities.

(c) ability to collect and record user statistics, thus monitoring the performance of the system.

Giving these three features highest priority, the development of the system could proceed along the following lines.

5.1 Compilation Time Error Messages

The provision at compile time of sufficiently clear and concise error messages, so that a user does not have to consult a special list to obtain their meaning. Developing this feature on the 1620 would require that the messages be stored on disk due to lack of core space. One technique would be to split each error message into separate phrases and then have each message represented by a sequence of pointers to the phrases that make up the message. This method would save space and allow for complex and sophisticated error messages.

Whether to print error messages as they occur within a program, or to wait until after the source program is listed is a debatable point. Printing them within the text with possibly a line with asterisks pinpointing the exact position of error makes it easy to see an error in a quick scan of the listing. On the other hand printing the messages after the source listing avoids spoiling the structure of the program. In MUSSEL's case I prefer the latter approach.

5.2 Execution-time Error Checking and Messages

The provision of numerous execution-time checks should be made to assist the programmer. The following should be checked at all times:

(i) Subscripts of subscripted variables out of range.

(ii) Reference to a variable before it has been assigned a value.

(iii) Division by zero and other arithmetic checks.

Checks such as these serve two purposes, execution-time errors are detected at the earliest possible moment, and also the system is safeguarded from being wrecked by programmers errors.
In MUSSEL, it would be a simple matter to pick up undefined variables as each variable has a typecode to check. Error messages printed should include the line number where the error occurred and provided the source program has been saved, as in MUSSEL's case, it is not too difficult to print out the source statement itself, where the error has occurred.

5.3 Symbol Table Dump at Execution Time

In interpretive systems it is a relatively easy matter to produce a symbolic dump of the Symbol Table during execution of a program. By providing extra space in the table you could also record the number of times a variable was referenced during the course of execution and have this listed as well. This would present no difficulty in MUSSEL as the names of identifiers have been stored on disk at the completion of the compilation phase.

5.4 Tracing a Program During Execution

This is an important and easy to implement facility that could be added to the system. By means of control cards at the head of a program could turn on this feature. Could have a card with $TRACE punched, which would cause a trace of all line numbers to be listed as program executes, or could have $TRACE ('list'), whereby at each subsequent execute-time reference to any name in the list, whose value changes printout the name, value of variable and line number where reference made.

5.5 Compile-time error diagnostics

As mentioned previously in section 4.3.12 the error diagnostics at compile-time are very rudimentary. It should be possible to improve this facility by changing the parsing from top-down to bottom-up when an error condition occurs. A reversed syntax-graph for use with a bottom-up algorithm could be stored on disk and brought into core when an error condition arises, together with a bottom-up analyser. It could also be possible to introduce some simple form of spelling correction, for example on instruction keywords.

5.6 Other Developments

With the advent of the Burrough's B6700 Computer at Massey University, one can envisage a much more sophisticated version of MUSSEL being implemented. It may be possible to interpret the source language directly, without the generation of Intermediate Language.

Taking an entirely different approach, maybe we could have an "Interactive MUSSEL" using Visual Display Units, whereby a MUSSEL program could be built up using the syntax of the language as explained by Hansen [27]. The structure of MUSSEL would lend itself to this form of treatment.

BIBLIOGRAPHY

1. Randell,B.,and Russell,L.J. ALGOL 60 Implementation. Academic Press, New York, 1964.
2. Gordon, N.M., and Gibbons, P.B. The Design and Implementation of a Structured Teaching Language.Massey University Computer Unit, Publication No.8, May 1972.
3. Dijkstra, E.W. Structured Programming Section 7.4, Software Engineering Techniques, Nato Science Committee, October 1969.
4. Dijkstra, E.W. Programming Considered as a Human Activity. Proceedings of IFIP Congress, 1965 Pp213-217.
5. Barton, R.S. Structured Programming - The Elimination of Branches. Software Engineering Coins III, Vol I, Pp.12
6. Wirth, N. Programming and Programming Languages. Presented to the International Computing Symposium, Bonn, May 1970.
7. Wirth, N. Program Development by Stepwise Refinement. C.A.C.M. Vol.14, No.4, April 1971.
8. Wirth, N. The Programming Language Pascal. Acta Informatica 1,35-63, 1971
9. Wirth, N. The Design of a PASCAL Compiler. Software-Practice and Experience, Vol.1, 309-333,1971
10. Doran, R.W. Structural Programming. Computer Education, No.9, October 1971
11. Doran, R.W. Teaching Structured Programming using an Unstructured Programming Language. Proceedings 3rd N.Z.Computer Conference, Vol.2, Pp.65-77, August 1972.
12. Doran, R.W., and Tate, G. An Approach to Structured Programming. Massey University Computer Unit, Publication No.6, 1972.
13. Henderson, P., and Snowdon,R. An Experiment in Structured Programming. Technical Report Series, No.18, University of Newcastle upon Tyne, 1971.

cont'd

14. Cohen, Doron J., and Gotlieb, C.C. A List Structure Form of Grammars for Syntactic Analysis. Computing Surveys, Vol.2, No 1, March 1970.

15. Wirth,N., and Weber, H. EULER: A Generalisation of ALGOL and its Formal Definition Parts I and II. C.A.C.M. Vol.9, Nos.1. and 2., 1966.

16. Morris, R. Scatter Storage Techniques. C.A.C.M. Vol.11, No.1, 1968.

17. Ledgard, Henry F. Ten Mini-Languages: A Study of Topical Issues in Programming Languages. Computing Surveys, Vol.3, No.3, September 1971.

18. Feldman,J., and Gries,D. Translator Writing Systems C.A.C.M. Vol.11, No.2, 1968.

19. Gries, D. Compiler Construction for Digital Computers John Wiley and Sons, 1971.

20. Cox, B.G. On the Teaching of Programming at Universities Proceedings 3rd N.Z. Computer Conference, Pp.79-90.

21. Hopgood, F.R.A. Compiling Techniques. Computer Monographs, No.8, MacDonald/Elsevier (4th ed.,1970)

22. Higman, B. A Comparitive Study of Programming Languages. Computer Monographs No.2, MacDonald/Elsevier (4th ed.,1970)

23. Glass, R.L. An Elementary Discussion of Compiler/Interpreter Writing. Computing Surveys, Vol.1, No.1, March 1969.

24. Knuth, D.E. and Floyd,R.W. Notes on Avoiding "GO TO" Statements. Information Processing Letters 1(1971) 23-31, North-Holland Publishing Company.

25. Conway, R.W. and Maxwell W.L. CORC - The Cornell Computing Language, CACM, vol 6, June, 1963 Pp317-321.

26, Conway,Melvin E., Design of a Separable Transition Diagram Compiler, CACM, vol 6, July, 1963 Pp 396-408

27. Hansen, Wilfred J., Creation of Hierarchic Text With a Computer Display, Argonne National Laboratory, Argonne,Illinois,1971

28. Cheathem,T.E., and Sattley,K. Syntax-directed Compiling. Proc. AFIPS 1964 Spring Joint Comput.Conf., Vol.25,Pp31-57.

APPENDIX A

THE SYNTAX OF MUSSEL IN BNF

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<logical value> ::= TRUE|FALSE|T|F

<delimiter> ::= <operator>|<bracket>|<separator>|<key>

<operator> ::= <arithmetic operator>|<string operator>|
<relational operator>|<logical operator<|<sequential
operator>

<arithmetic operator> ::= +|-|*|/|./.|**

<string operator> ::= .CAT.

<relational operator> ::= .LT.|.LE.|.EQ.|.NE.|.GE.|.GT.

<logical operator> ::= .OR.|.AND.|.NOT.

<sequential operator> ::= THEN|ELSE

<bracket> ::= (|)|!|DO|END

<separator> ::= ,|.|:|TO| |UNTIL|WHILE|FROM|BY|ON|AS|IN|IS|FOR|TIMES

<key> ::= RES|RESERVE|SET|READ|PRINT|EXIT|REPEAT|CASE|IF|CHOICE|DEFINE|
VALUE|EXECUTE

<PROGRAM> ::= <group begin> <eol> <reserve> <instructions> <group end>

<instruction> ::= <simple instruction>|<group instruction>|<label> :
<group instruction>

<simple instruction> ::= <reserve>|<assign>|<read>|<print>|<exit>|
<subroutine definition>|<subroutine execute>|
<function value assign>

<group instruction> ::= <simple group>|<repeat group>|<choice group>
|<case group>|<conditional group>|<if group>

<reserve> ::= RESERVE <loclist>|RES <loclist>

<loclist> ::= <simple variable>|<arrayspec>|<multiple arrayspec>
|<loclist>,<simple variable>|<loclist>,<arrayspec>|
<loclist>, <multiple arrayspec>

<simple variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<identifier> ::= <letter>|<identifier> <letter>|<identifier> <digit>

<arrayspec> ::= <array identifier> (<bound pair list>)

<bound pair list> ::= <bound pair>|<bound pair>,<bound pair>

<bound pair> ::= <valexp> : <valexp>

<array identifier> ::= <identifier>

<multiple arrayspec> ::= (<array segment>)

<array segment> ::= <array list> ,<arrayspec>

<array list> ::= <array identifier>|<array list>,<array identifier>

<assign> ::= SET <list part> TO <expression>

<list part> ::= <variable>|<list part>,<variable>

<variable> ::= <simple variable>|<array variable>

<array variable> ::= <array identifier>(<valexp>)|<array identifier>
(<valexp>,<valexp>)|<array identifier>

<read> ::= READ <list part>

<print> ::= PRINT <print list>

<print list> ::= <print el>|<print control>|<print list>,<print el>|
<print list>,<print control>

<print el> ::= <expression> <picture control>|(<expression>,<print el>)

<picture control> ::= (PIC = <picture>)|<empty>

<picture> ::= <alpha picture>|<numeric picture>

<alpha picture> ::= <first alpha>|<alpha picture> <first alpha>|
<alpha picture>,<first alpha>|<alpha picture>
(<integer>)

<first alpha> ::= */B

<numeric picture> ::= <sign> <integer part> <decimal part>
<exponent part> <sign>

<sign> ::= -|S|<empty>

<integer part> ::= <firstnum>|<integer part> <firstnum>| <integer part>
<firstnum>|<integer part>(<integer>)|<empty>

<decimal part> ::= .<integer part>|<empty>

<exponent part> ::= E <sign>**| E <sign> 99| <empty>

<firstnum> ::= *|9|B

<empty> ::=

<print control> ::= NEWPAGE|NEWLINE|NEWLINE(<valexp>)|SPACE(<valexp>)|
TAB(<valexp>)

<exit> ::= EXIT|EXIT FROM <label>

<label> ::= <identifier>

<subroutine definition> ::= DEFINE<subroutine identifier> AS|
DEFINE <subroutine identifier> ON
<parameter list> AS

<subroutine identifier> ::= <identifier>

<parameter list> ::= <parameter identifier >|<parameter list>,
<parameter identifier>

<parameter identifier> ::= <identifier>

<subroutine execute> ::= EXECUTE <subroutine identifier>|EXECUTE
<subroutine identifier>(<exp list>)

<exp list> ::= <expression>|<exp list>,<expression>

<function value assign> ::= VALUE IS <expression>

<group begin> ::= DO|<label> : DO

<simple group> ::= <group begin> <eol> <instructions> <group end>

<instructions> ::= <instruction>|<instructions> <instruction>

<group end> ::= END <eol>|END <label> <eol>

```
<repeat group> ::= <group begin> REPEAT <eol> <instructions>
                   <group end>|<group begin> REPEAT <control> <eol>
                   <instructions> <group end>

<control> ::= UNTIL <condexp>| WHILE <condexp>|<valexp> TIMES|FOR
              <variable> FROM <valexp> <for end >|FOR <variable>
              SET TO <explist>

<for end> ::= TO <valexp> BY <valexp>| BY <valexp>|TO <valexp>
              |<empty>

<choice group> ::= <group begin> CHOICE OF <eol> <if group>
                   <group end>

<case group> ::= <case head> <instructions> <group end>

<case head> ::= <group begin> CASE <valexp> OF <eol>|<group begin>
                CASE <valexp> IN (<integer>,<integer>) OF <eol>

<if group> ::= IF <condexp> THEN <simple instruction>| IF <condexp>
               <eol> THEN <instruction>

<conditional group> ::= <group begin> IF <condexp> <eol> THEN
                        <instruction> <group end>|<group begin>
                        IF <condexp> <eol> THEN <instruction> ELSE
                        <instruction> <group end>

<expression> ::= <valexp>|<condexp>|<stringexp>

<valexp> ::= <term>|<adding op> <term>|<valexp> <adding op> <term>

<term> ::= <factor>|<term> <mult op > <factor>

<factor> ::= <primary>|<factor> ** <primary>

<primary> ::= <unsigned number>|<variable>|<function call>|(<valexp>)

<mult op> ::= * |/|./.

<adding op> ::= +|-

<condexp> ::= <logical term>|<condexp> .OR. <logical term>

<logical term> ::= <logical factor>|<logical term> .AND. <logical
                   factor>

<logical factor> ::= <logical primary>| .NOT. <logical primary>
```

<logical primary> ::= <logical value>|<variable>|<function call>|
<relation>|(<condexp>)

<relation> ::= <valexp> <relational op > <value exp>|<stringexp>
<relational op> <string exp>

<string exp> ::= <string term>|<string exp> .CAT. <string term>

<string term> ::= <variable>|<string>|<function call>|<string exp>

<function call> ::= <function identifier>|<function identifier >
(<explist>)

<function identifier> ::= <identifier>|<library function>

<library function> ::= LOG|EXP|ABS|SIN|COS|SQRT|STRING|NUMBER|INTEGER|
CONDITION|LENGTH|POSITION|SUBSTRING|COUNT|
MAXIMUM|MINIMUM

<number> ::= <unsigned number>| + <unsigned number>| - <unsigned number>

<unsigned number> ::= <decimal number> <exponent>|<decimal number>

<decimal number> ::= <unsigned integer>|<decimal fraction>|
<unsigned integer> <decimal fraction>

<exponent> ::= E <integer>

<decimal fraction> ::= .<unsigned integer>

<integer> ::= <unsigned integer> | + <unsigned integer>|
- <unsigned integer>

<unsigned integer> ::= <digit>|<unsigned integer> <digit>

<proper string> ::= <string character>|<proper string> <string
character>

<string character> ::= <letter>|<digit>|<quote>|<special character>|
<empty>

<quote> ::= !!

<special character> ::= :|.|*|+|-|/|$|@|≠|)|(

<string> ::= ! <proper string> !

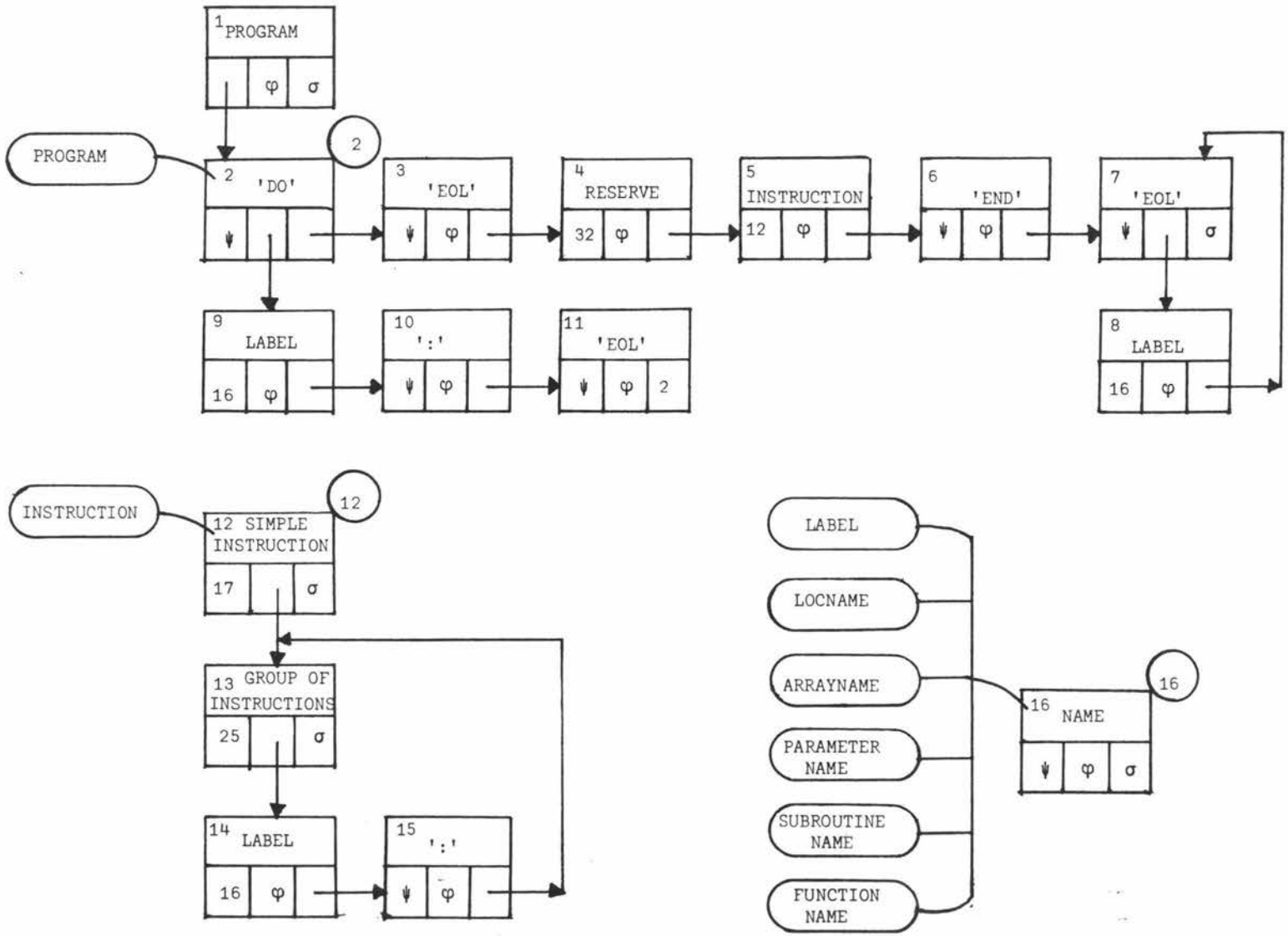
<eol> ::= end of line - not represented externally.

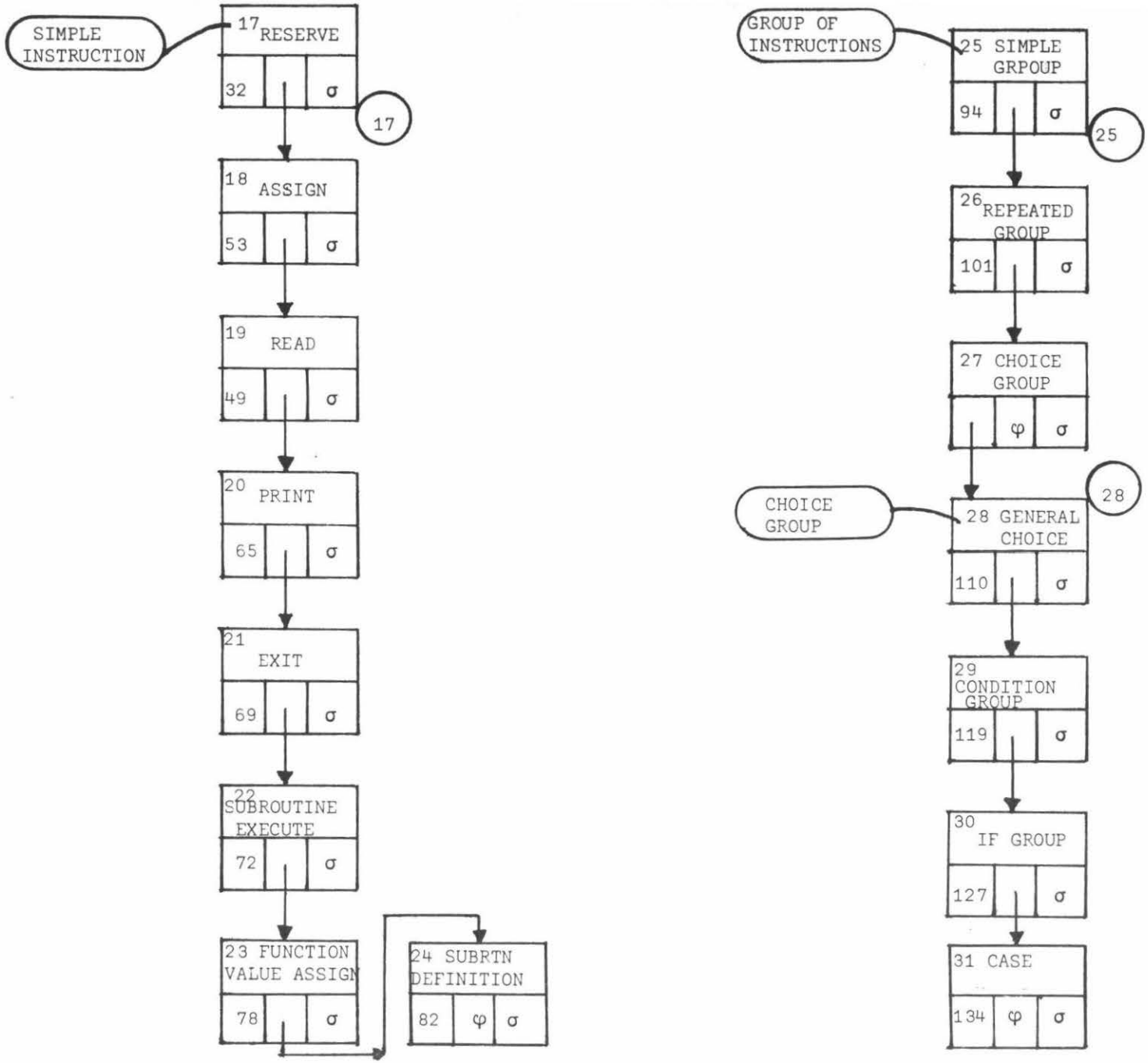
APPENDIX B

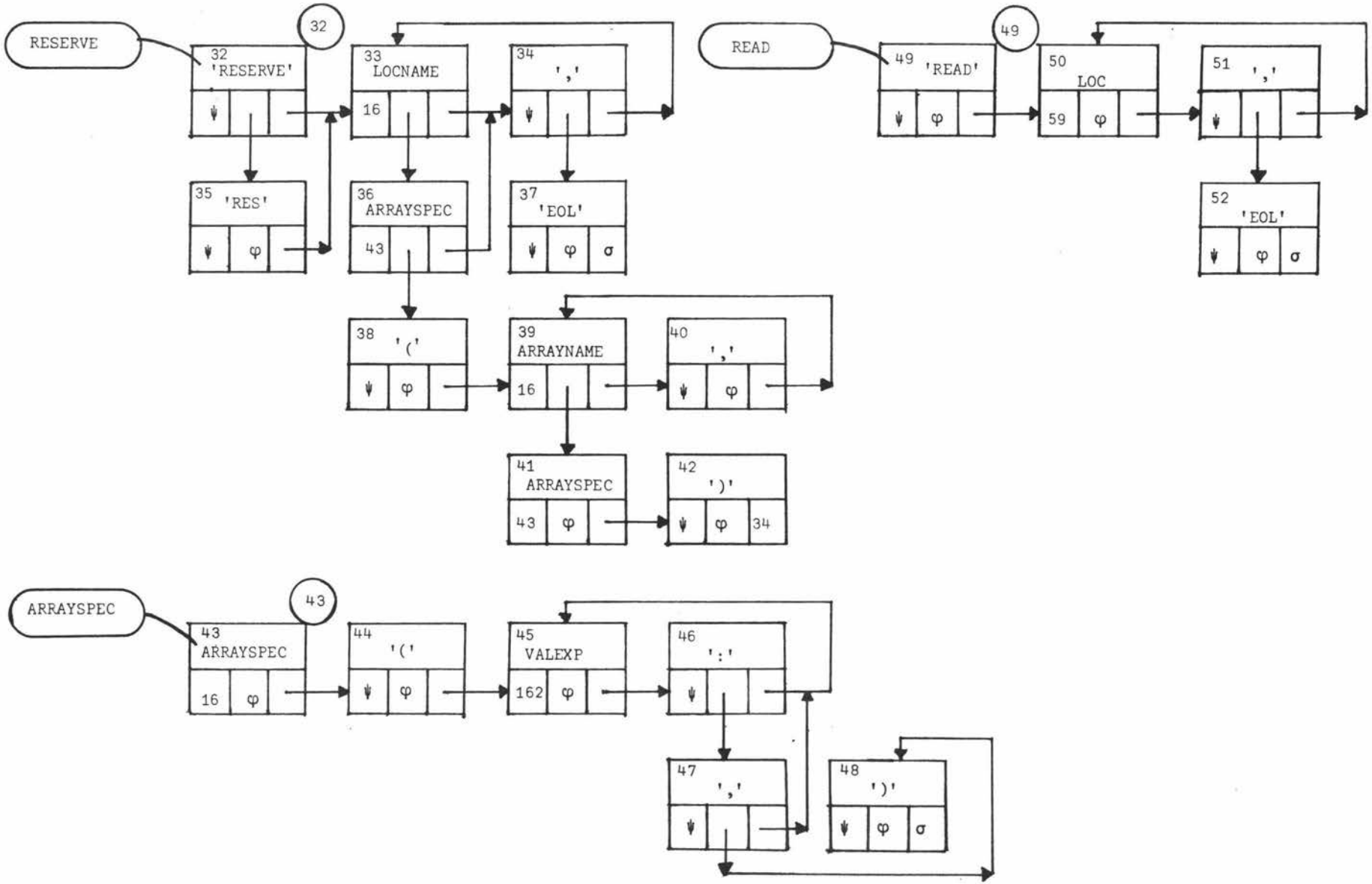
REDUCED SYNTAX GRAPH FOR

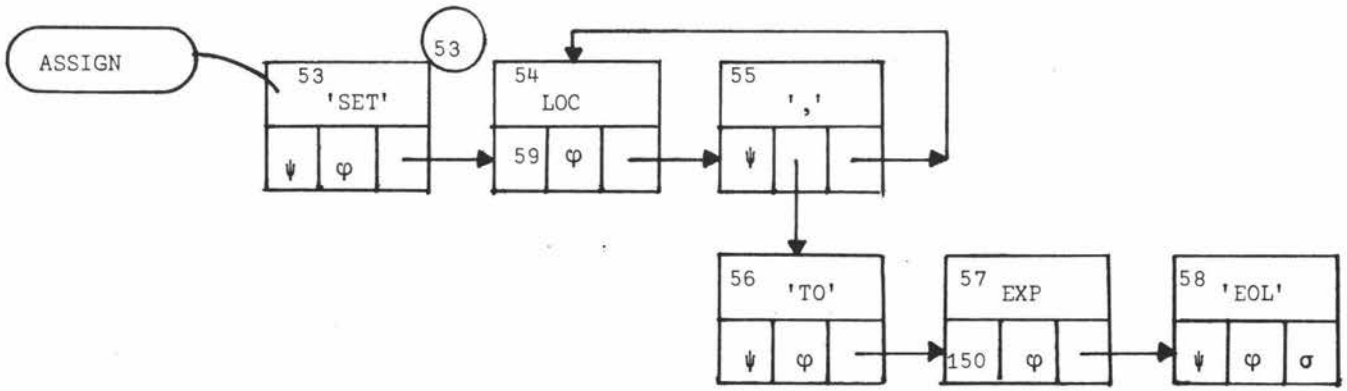
THE MUSSEL GRAMMAR

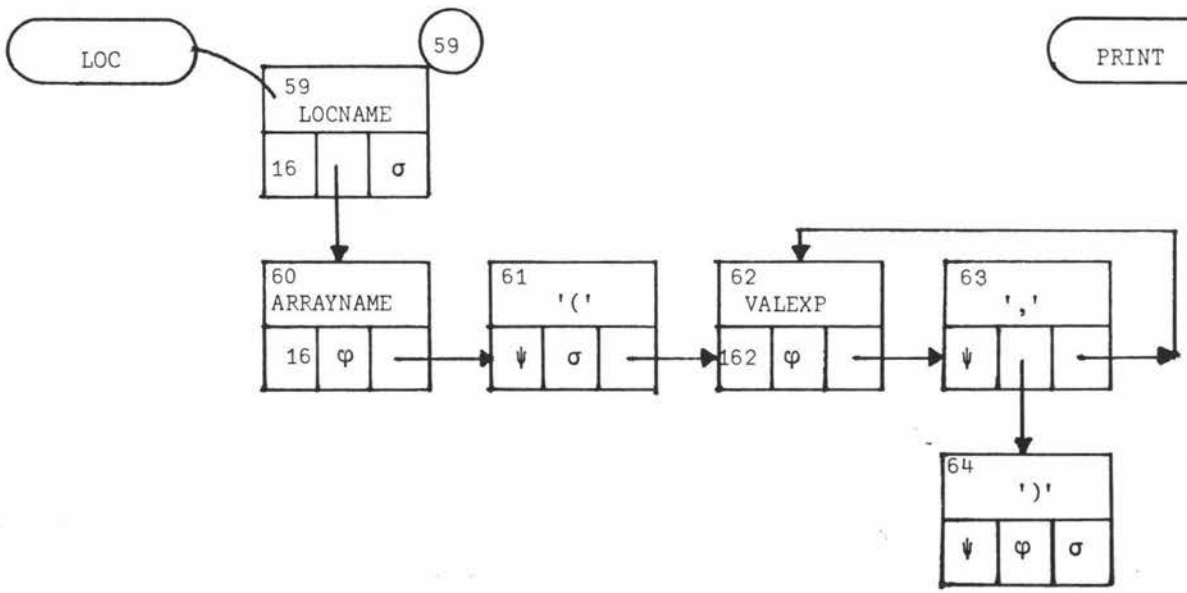
Note The DEF link for terminal nodes is denoted by $\Downarrow$
The ALT link for last alternative in a production is denoted by φ. ALT links denoted by σ are "dont care" conditions.
The SUC link of the last component of each production is denoted by σ.

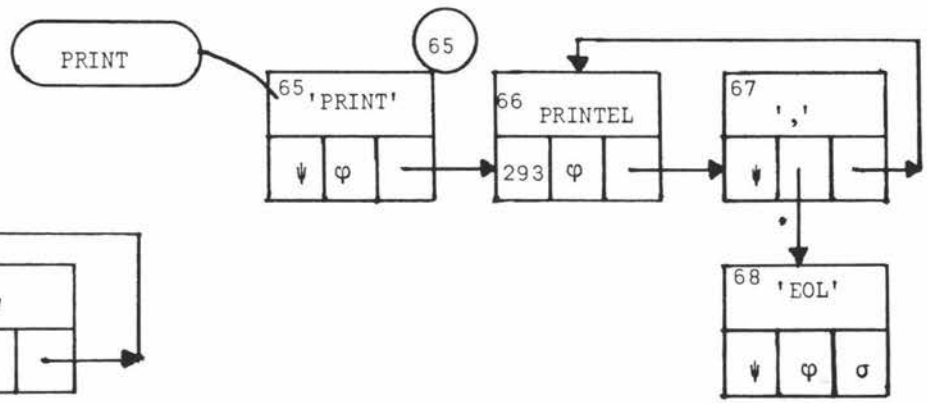

1 PROGRAM
φ
σ
PROGRAM
2
2 'DO'
3 'EOL'
4 RESERVE
32
5 INSTRUCTION
12
6 'END'
7 'EOL'
9 LABEL
16
10 ':'
11 'EOL'
2
8 LABEL
16
INSTRUCTION
12
12 SIMPLE INSTRUCTION
17
13 GROUP OF INSTRUCTIONS
25
14 LABEL
16
15 ':'
LABEL
LOCNAME
ARRAYNAME
PARAMETER NAME
SUBROUTINE NAME
FUNCTION NAME
16 NAME

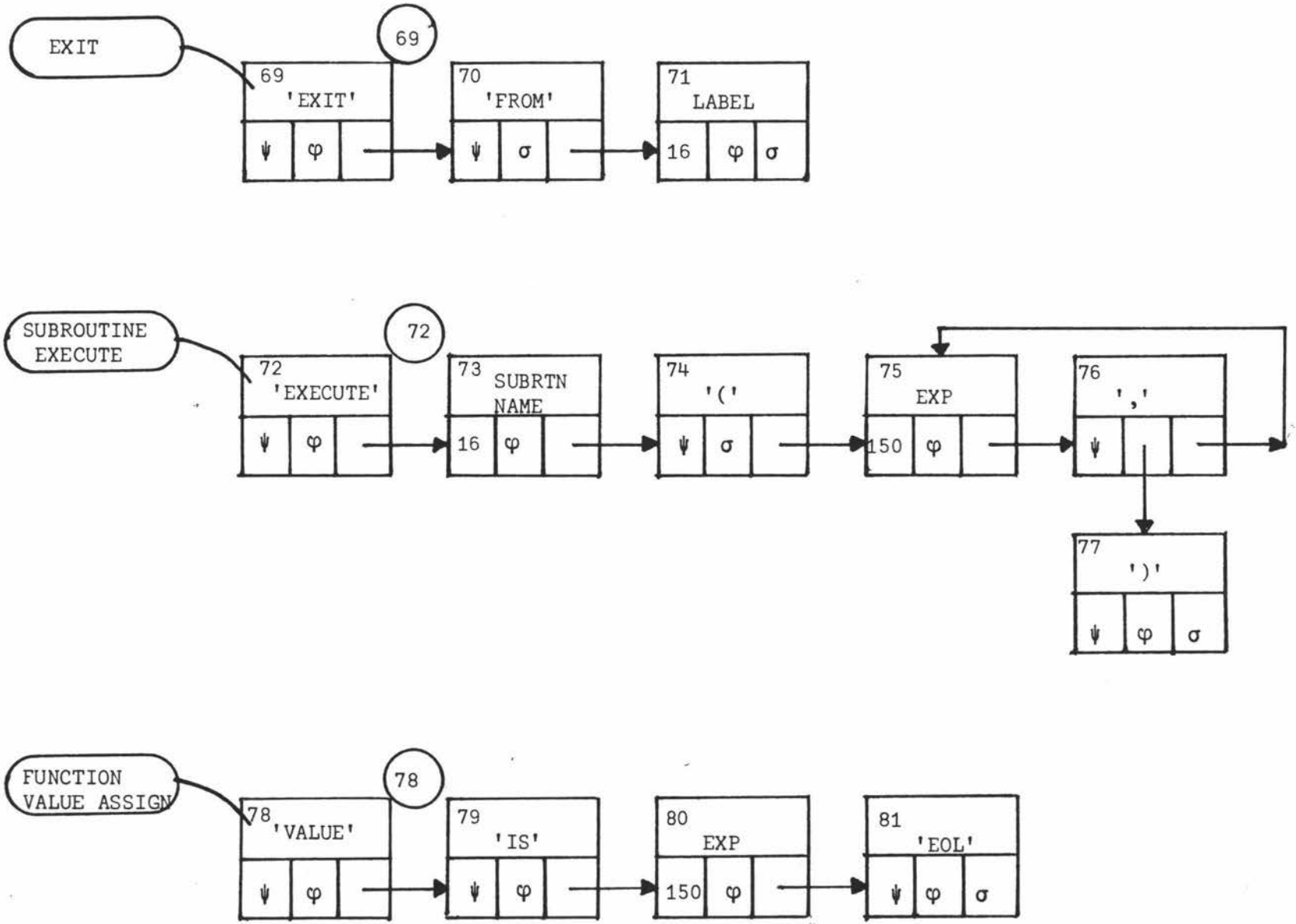

SIMPLE INSTRUCTION
17 RESERVE
32 σ
17
18 ASSIGN
53 σ
19 READ
49 σ
20 PRINT
65 σ
21 EXIT
69 σ
22 SUBROUTINE EXECUTE
72 σ
23 FUNCTION VALUE ASSIGN
78 σ
24 SUBRTN DEFINITION
82 φ σ
GROUP OF INSTRUCTIONS
25 SIMPLE GRPOUP
94 σ
25
26 REPEATED GROUP
101 σ
27 CHOICE GROUP
φ σ
CHOICE GROUP
28 GENERAL CHOICE
110 σ
28
29 CONDITION GROUP
119 σ
30 IF GROUP
127 σ
31 CASE
134 φ σ

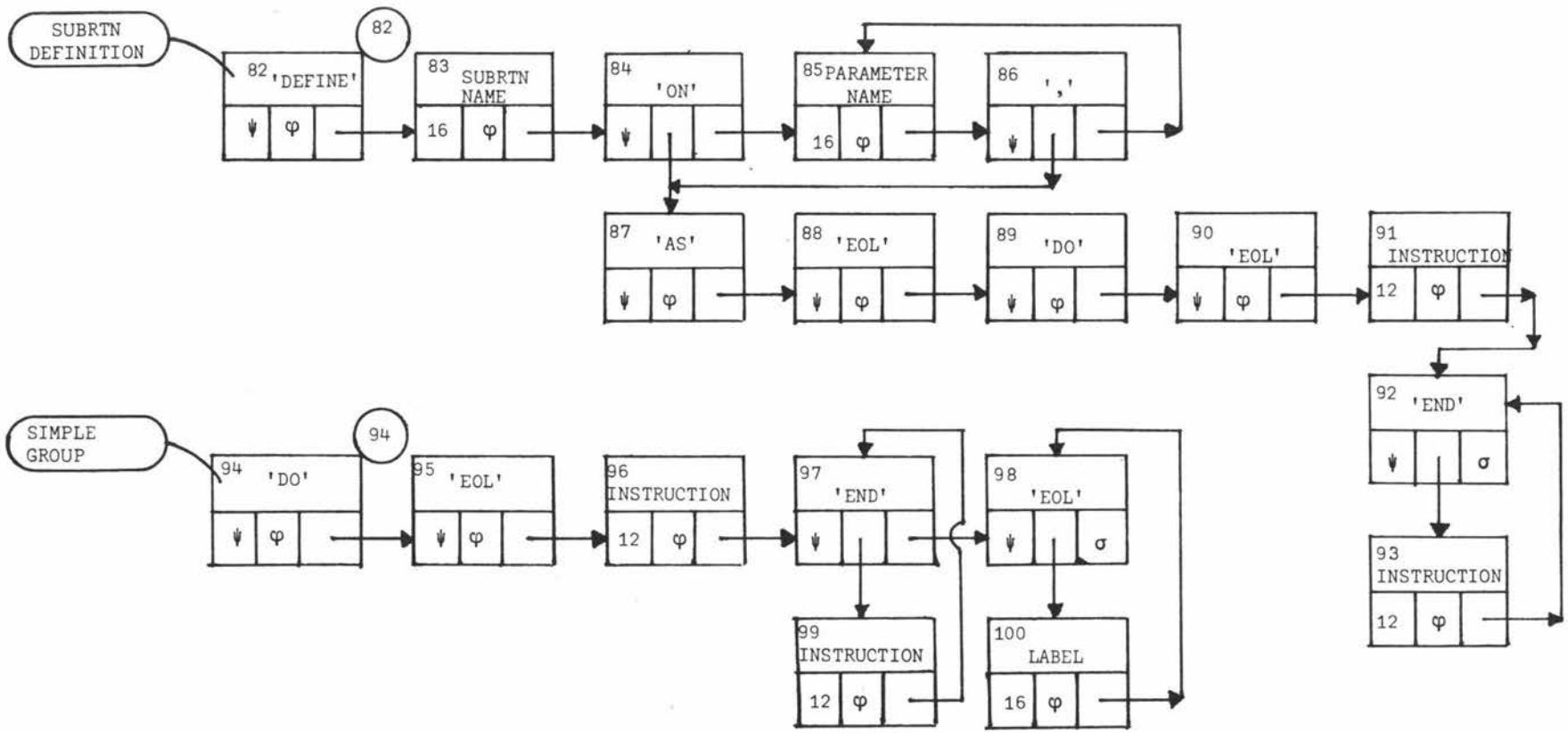

RESERVE
32
32 'RESERVE'
33 LOCNAME
16
34 ','
35 'RES'
36 ARRAYSPEC
43
37 'EOL'
38 '('
39 ARRAYNAME
16
40 ','
41 ARRAYSPEC
43
42 ')'
34
READ
49
49 'READ'
50 LOC
59
51 ','
52 'EOL'
ARRAYSPEC
43
43 ARRAYSPEC
16
44 '('
45 VALEXP
162
46 ':'
47 ','
48 ')'

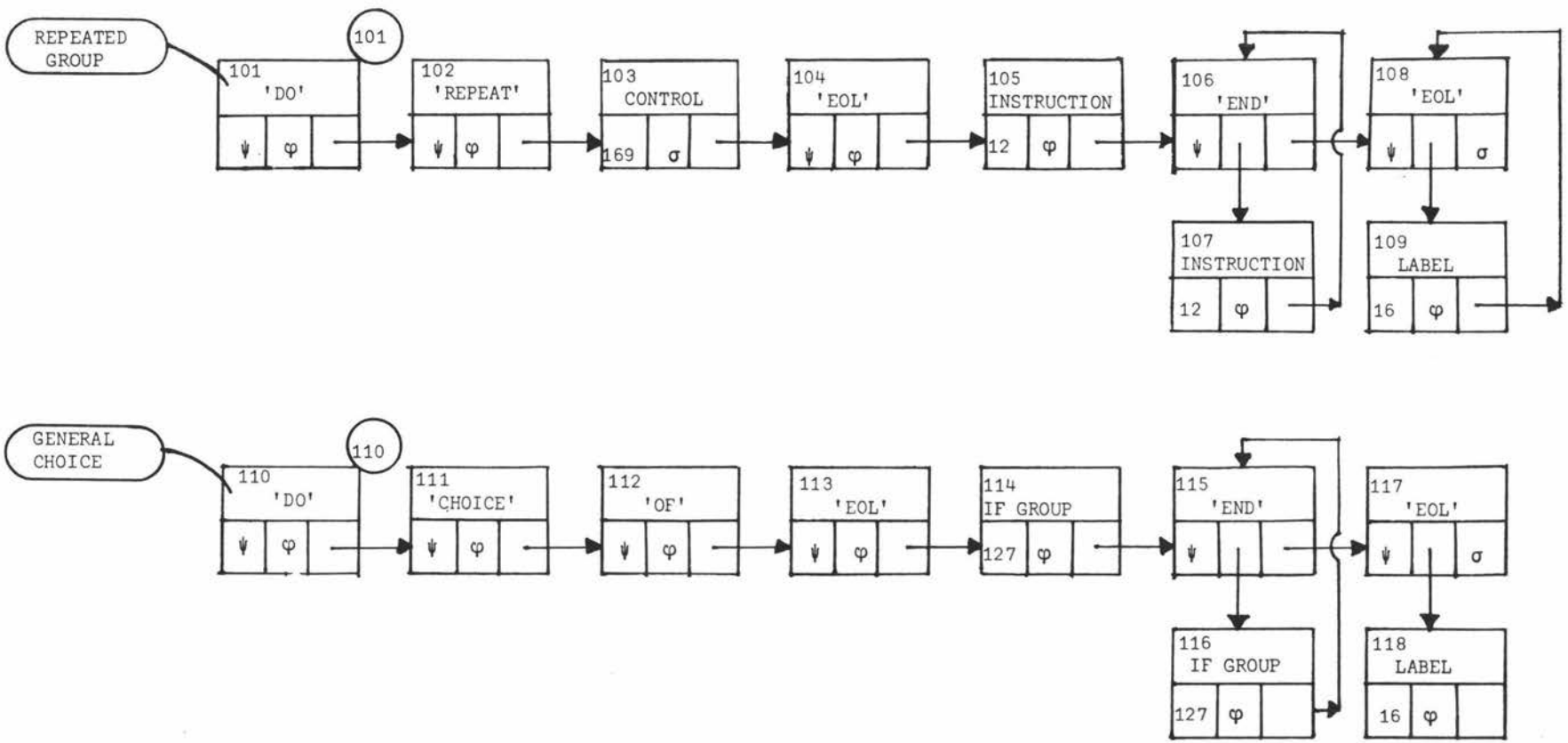

ASSIGN
53
53 'SET'
54 LOC
59
55 ','
56 'TO'
57 EXP
150
58 'EOL'

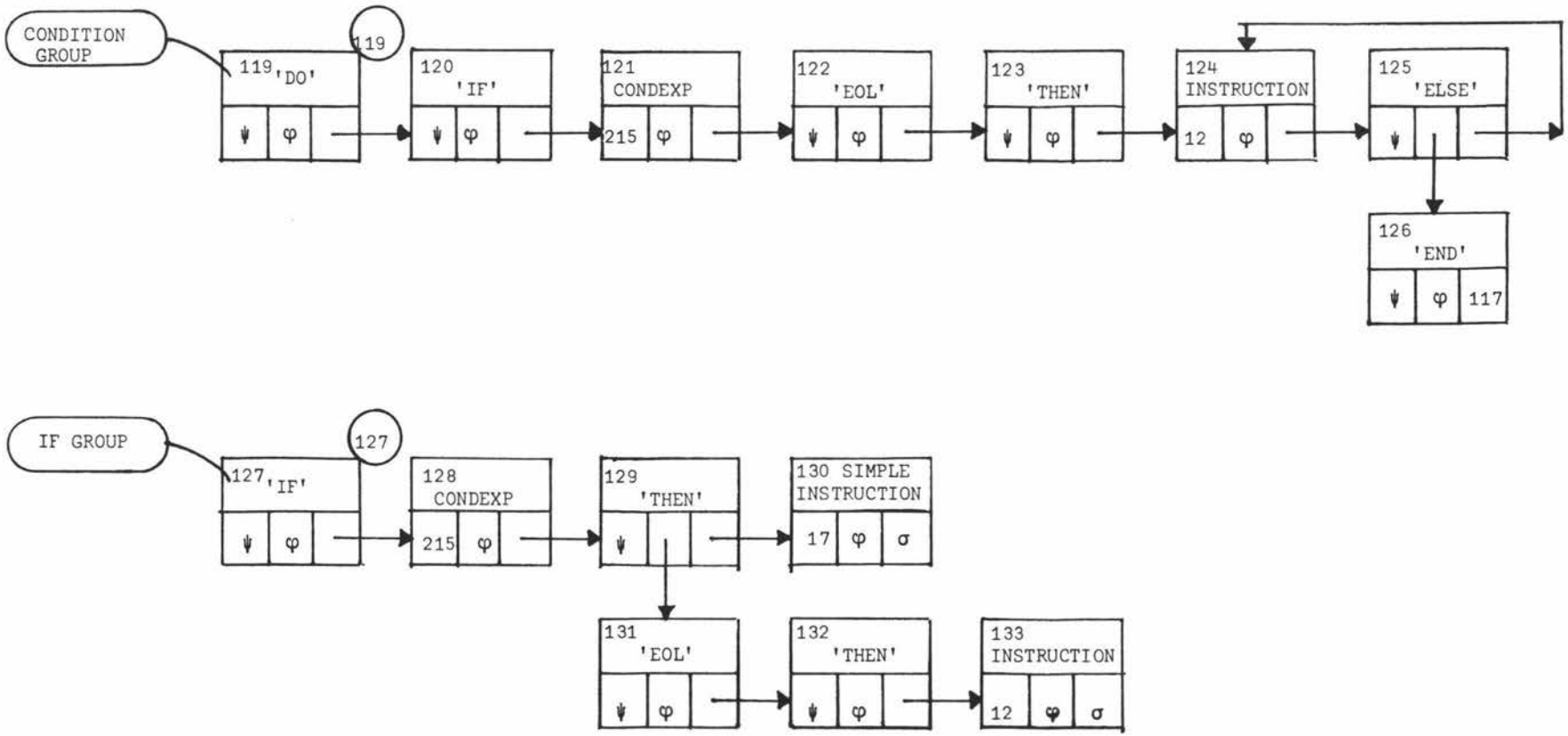

LOC
59
59 LOCNAME
16
60 ARRAYNAME
16
61 '('
62 VALEXP
162
63 ','
64 ')'

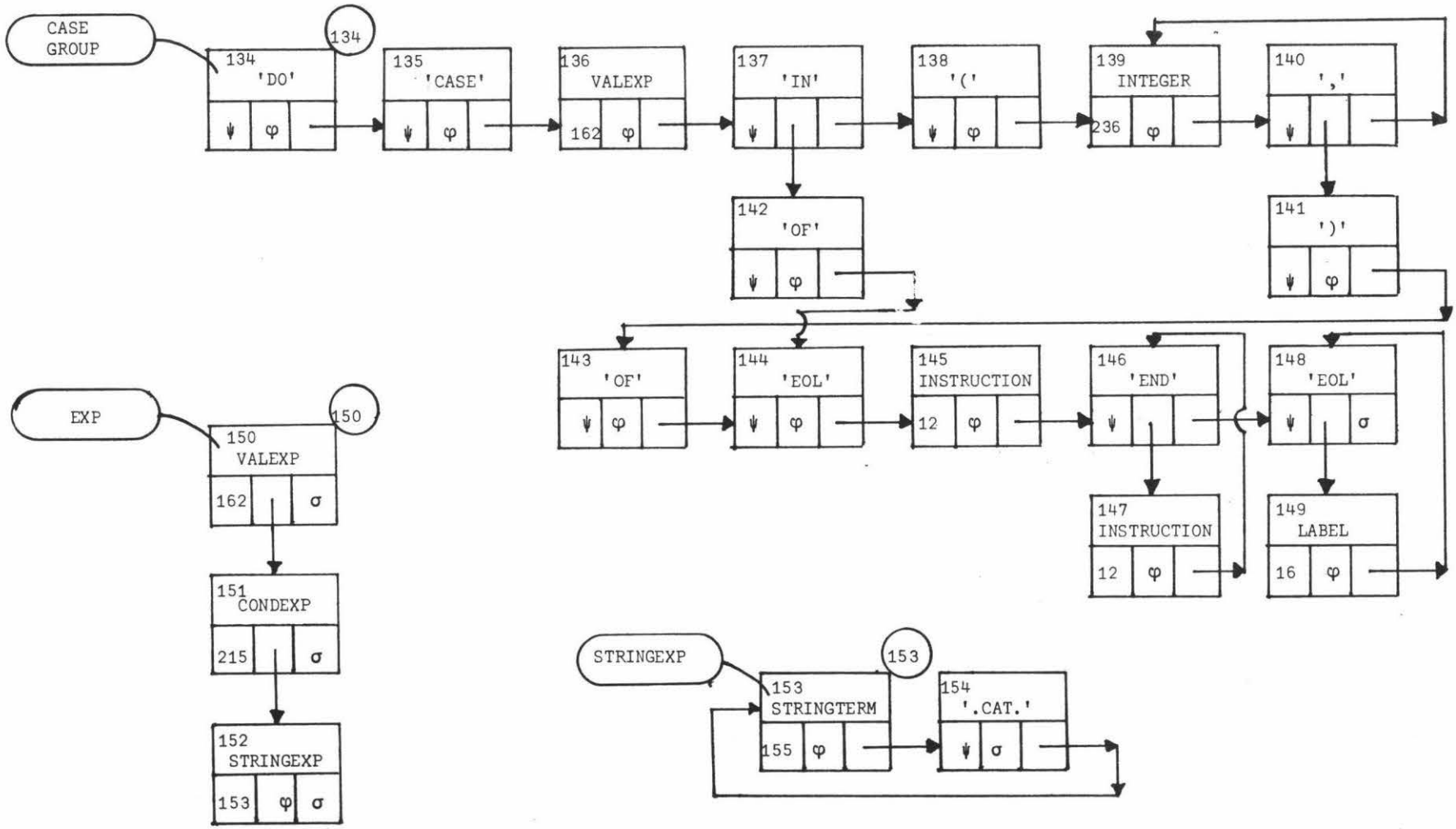

PRINT
65
65 'PRINT'
66 PRINTEL
293
67 ','
68 'EOL'

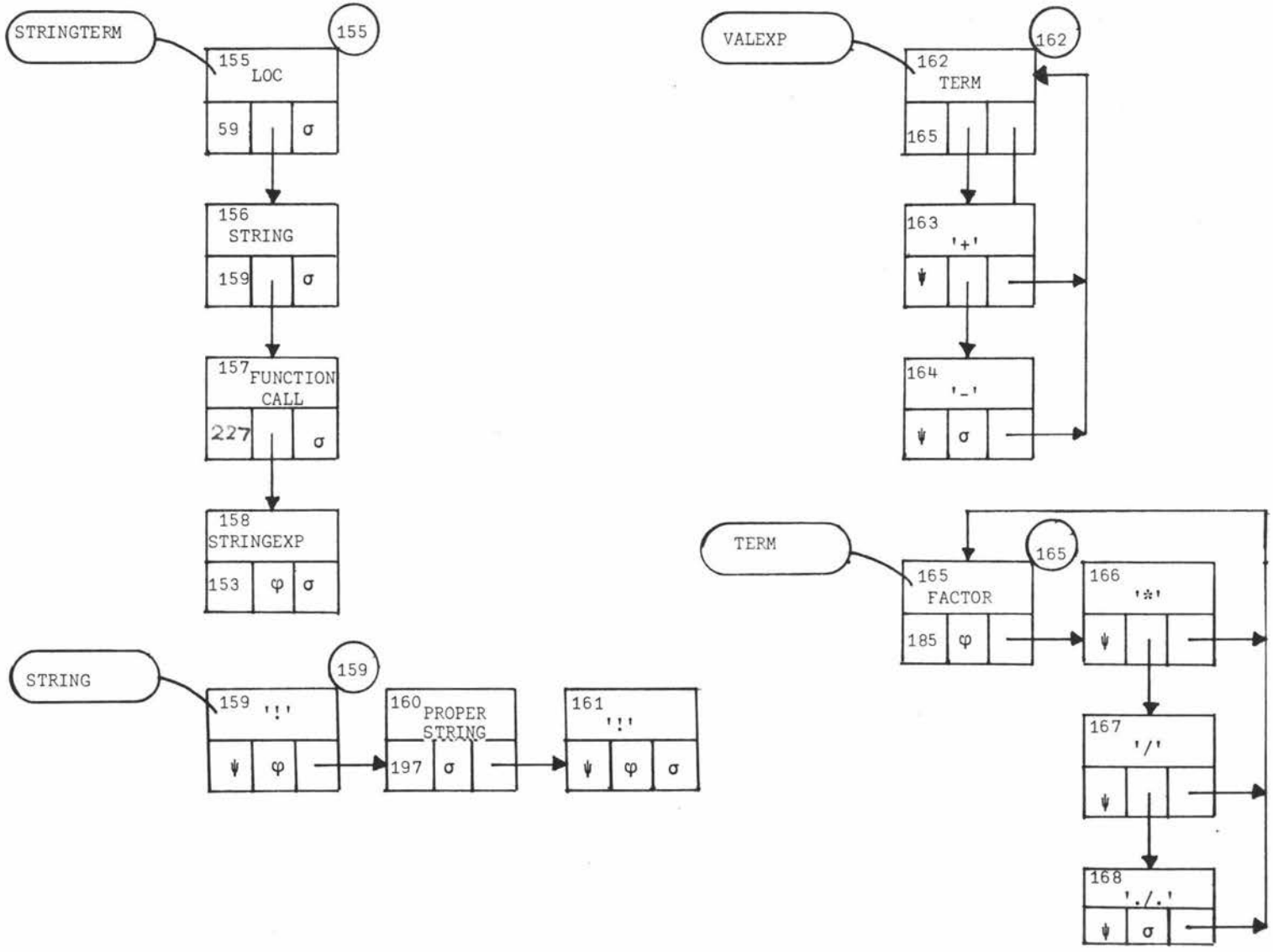

EXIT
69
69
'EXIT'
ψ
φ
70
'FROM'
ψ
σ
71
LABEL
16
φ
σ
SUBROUTINE
EXECUTE
72
72
'EXECUTE'
ψ
φ
73
SUBRTN
NAME
16
φ
74
'('
ψ
σ
75
EXP
150
φ
76
','
ψ
77
')'
ψ
φ
σ
FUNCTION
VALUE ASSIGN
78
78
'VALUE'
ψ
φ
79
'IS'
ψ
φ
80
EXP
150
φ
81
'EOL'
ψ
φ
σ

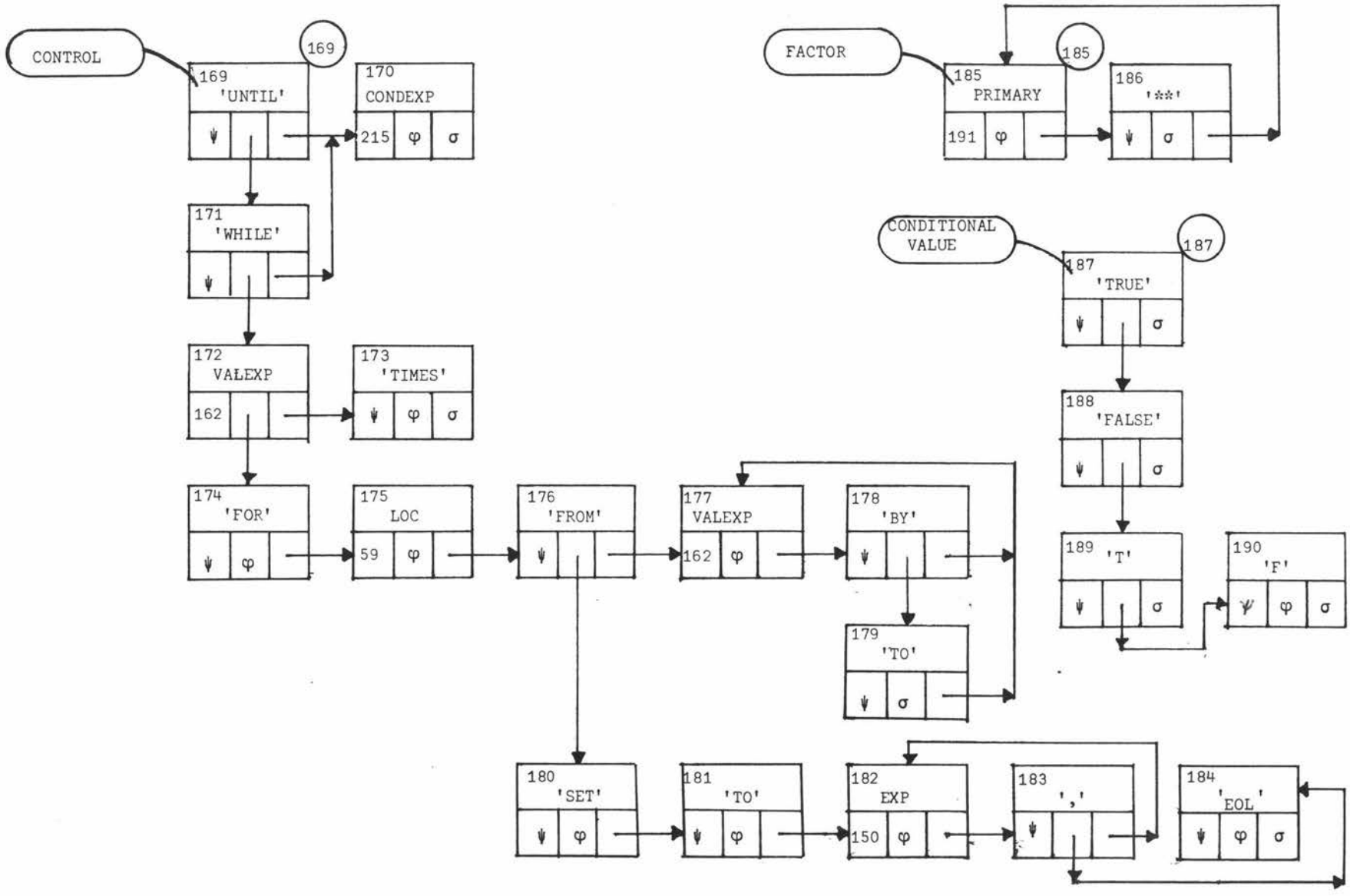

SUBRTN DEFINITION
82
82 'DEFINE'
83 SUBRTN NAME
16
84 'ON'
85 PARAMETER NAME
16
86 ','
87 'AS'
88 'EOL'
89 'DO'
90 'EOL'
91 INSTRUCTION
12
92 'END'
σ
93 INSTRUCTION
12
SIMPLE GROUP
94
94 'DO'
95 'EOL'
96 INSTRUCTION
12
97 'END'
98 'EOL'
σ
99 INSTRUCTION
12
100 LABEL
16

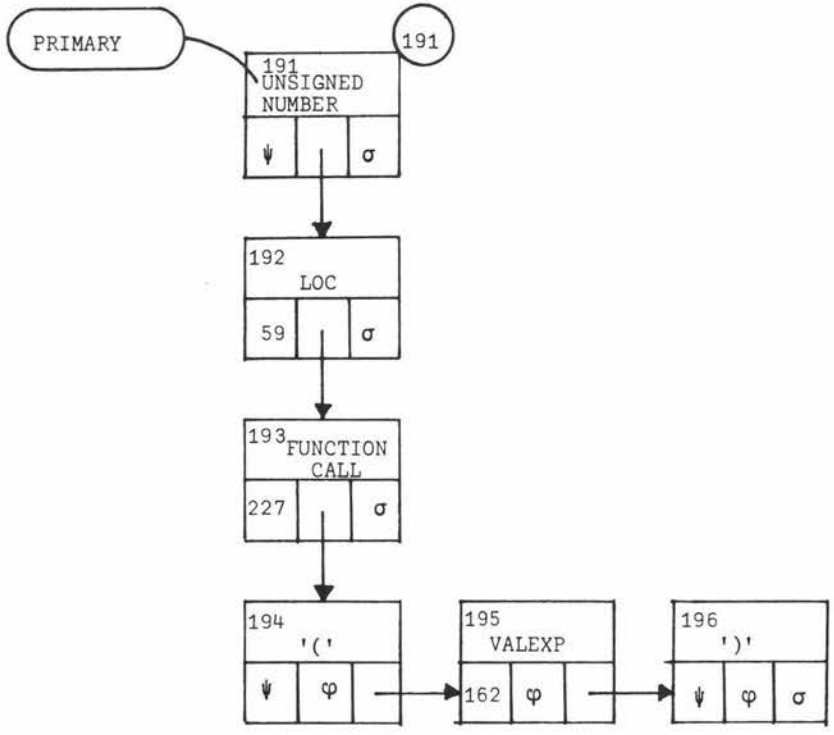

REPEATED GROUP
101
101 'DO'
102 'REPEAT'
103 CONTROL
169 σ
104 'EOL'
105 INSTRUCTION
12 φ
106 'END'
107 INSTRUCTION
12 φ
108 'EOL'
109 LABEL
16 φ
GENERAL CHOICE
110
110 'DO'
111 'CHOICE'
112 'OF'
113 'EOL'
114 IF GROUP
127 φ
115 'END'
116 IF GROUP
127 φ
117 'EOL'
118 LABEL
16 φ

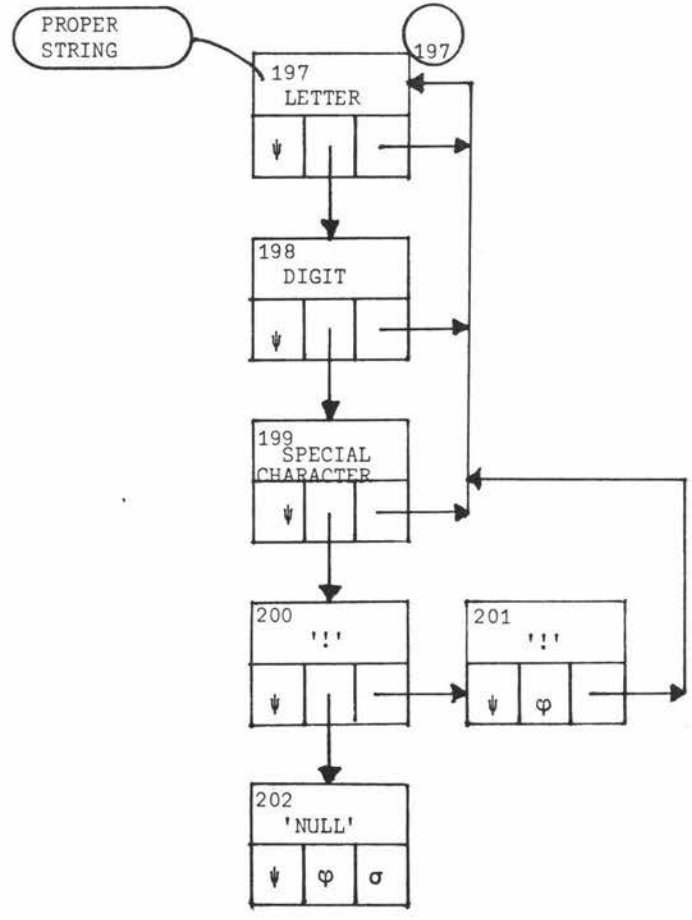

CONDITION GROUP
119
119 'DO'
120 'IF'
121 CONDEXP
215
122 'EOL'
123 'THEN'
124 INSTRUCTION
12
125 'ELSE'
126 'END'
117
IF GROUP
127
127 'IF'
128 CONDEXP
215
129 'THEN'
130 SIMPLE INSTRUCTION
17
σ
131 'EOL'
132 'THEN'
133 INSTRUCTION
12
σ

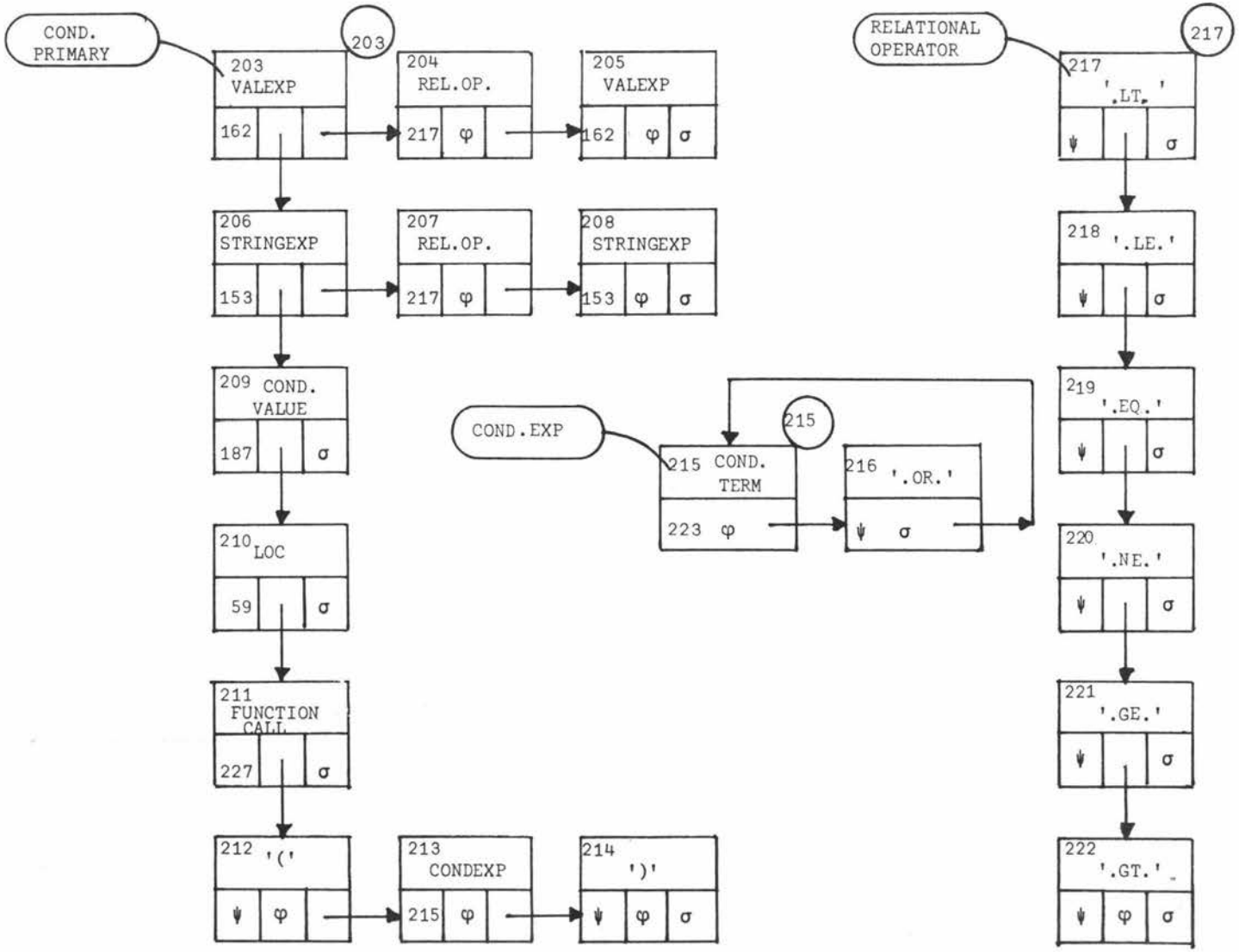

CASE GROUP
134
134 'DO'
135 'CASE'
136 VALEXP
162
137 'IN'
138 '('
139 INTEGER
236
140 ','
142 'OF'
141 ')'
143 'OF'
144 'EOL'
145 INSTRUCTION
12
146 'END'
148 'EOL'
147 INSTRUCTION
12
149 LABEL
16
EXP
150
150 VALEXP
162
151 CONDEXP
215
152 STRINGEXP
153
STRINGEXP
153
153 STRINGTERM
155
154 '.CAT.'

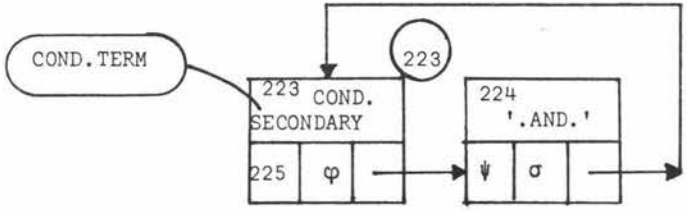

STRINGTERM
155
155
LOC
59
σ
156
STRING
159
σ
157
FUNCTION
CALL
227
σ
158
STRINGEXP
153
φ
σ
VALEXP
162
162
TERM
165
163
'+'
ψ
164
'-'
ψ
σ
TERM
165
165
FACTOR
185
φ
166
'*'
ψ
167
'/'
ψ
168
'./.'
ψ
σ
STRING
159
159
'!'
ψ
φ
160
PROPER
STRING
197
σ
161
'!'
ψ
φ
σ

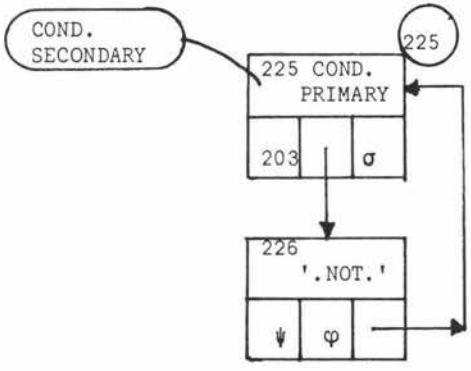

CONTROL
169
169
'UNTIL'
170
CONDEXP
215 φ σ
171
'WHILE'
172
VALEXP
162
173
'TIMES'
174
'FOR'
175
LOC
59 φ
176
'FROM'
177
VALEXP
162 φ
178
'BY'
179
'TO'
180
'SET'
181
'TO'
182
EXP
150 φ
183
','
184
'EOL'
FACTOR
185
185
PRIMARY
191 φ
186
'**'
CONDITIONAL
VALUE
187
187
'TRUE'
188
'FALSE'
189
'T'
190
'F'

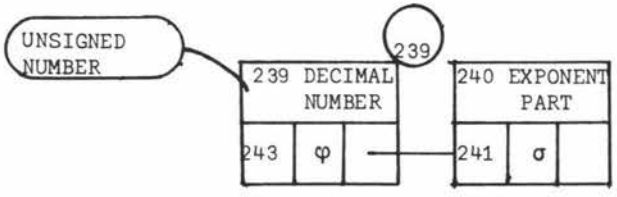

PRIMARY
191
191
UNSIGNED
NUMBER
192
LOC
59
193
FUNCTION
CALL
227
194
'('
195
VALEXP
162
196
')'

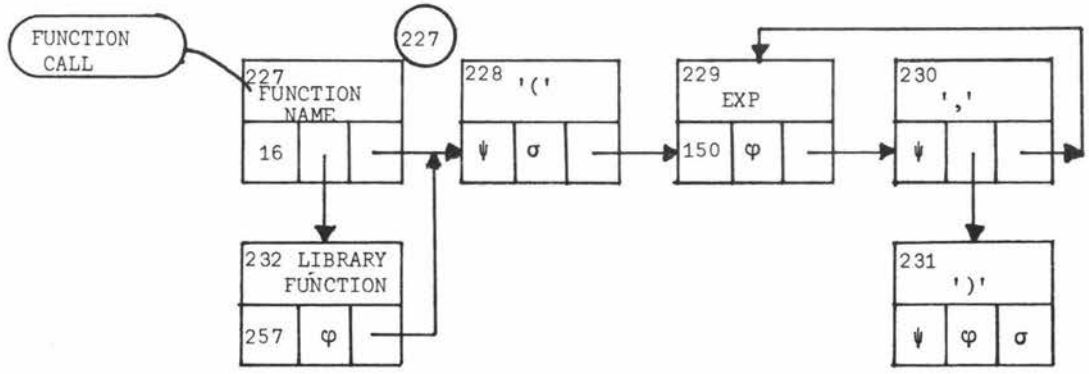

PROPER
STRING
197
197
LETTER
198
DIGIT
199
SPECIAL
CHARACTER
200
'!'
201
'!'
202
'NULL'

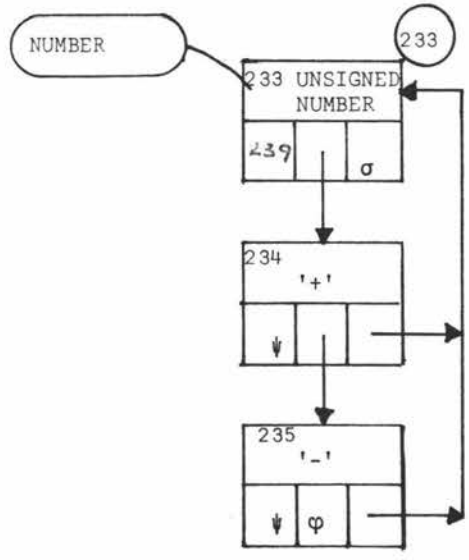

COND. PRIMARY
203
203 VALEXP
162
204 REL.OP.
217 φ
205 VALEXP
162 φ σ
206 STRINGEXP
153
207 REL.OP.
217 φ
208 STRINGEXP
153 φ σ
209 COND. VALUE
187 σ
210 LOC
59 σ
211 FUNCTION CALL
227 σ
212 '('
ψ φ
213 CONDEXP
215 φ
214 ')'
ψ φ σ
COND.EXP
215
215 COND. TERM
223 φ
216 '.OR.'
ψ σ
RELATIONAL OPERATOR
217
217 '.LT.'
ψ σ
218 '.LE.'
ψ σ
219 '.EQ.'
ψ σ
220 '.NE.'
ψ σ
221 '.GE.'
ψ σ
222 '.GT.'
ψ φ σ

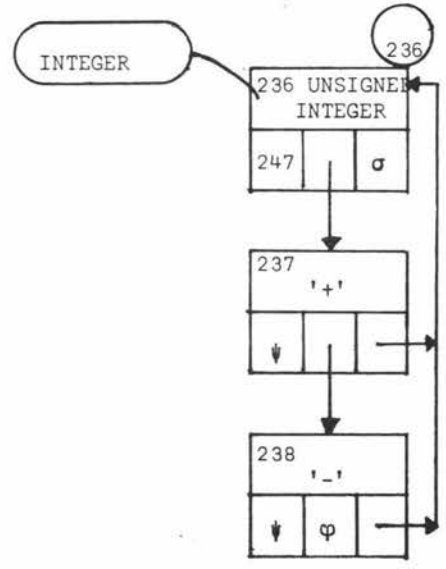

COND.TERM
223
223 COND. SECONDARY
225 φ
224 '.AND.'
ψ σ

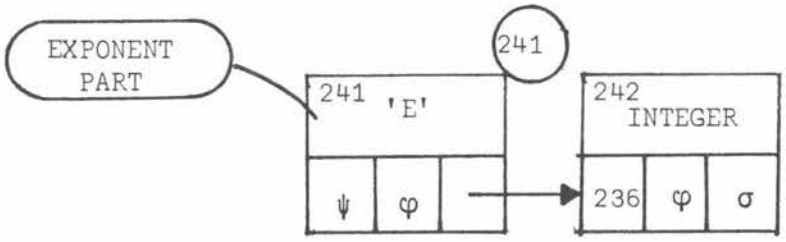

COND. SECONDARY
225
225 COND. PRIMARY
203 σ
226 '.NOT.'
ψ φ

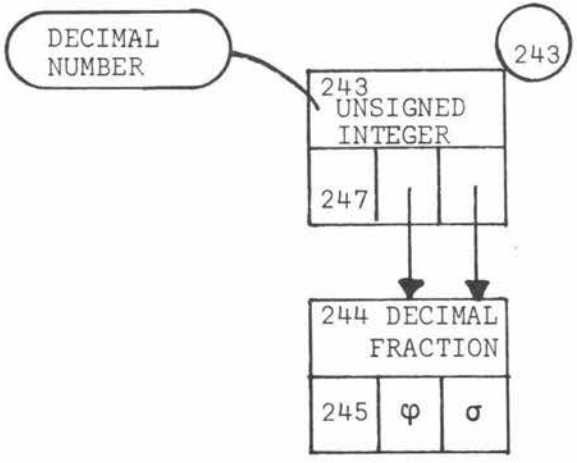

UNSIGNED NUMBER
239
239 DECIMAL NUMBER
243 φ
240 EXPONENT PART
241 σ

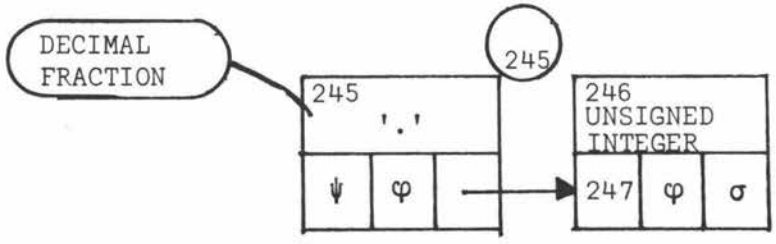

FUNCTION CALL
227
227 FUNCTION NAME
16
228 '('
ψ σ
229 EXP
150 φ
230 ','
ψ
232 LIBRARY FUNCTION
257 φ
231 ')'
ψ φ σ

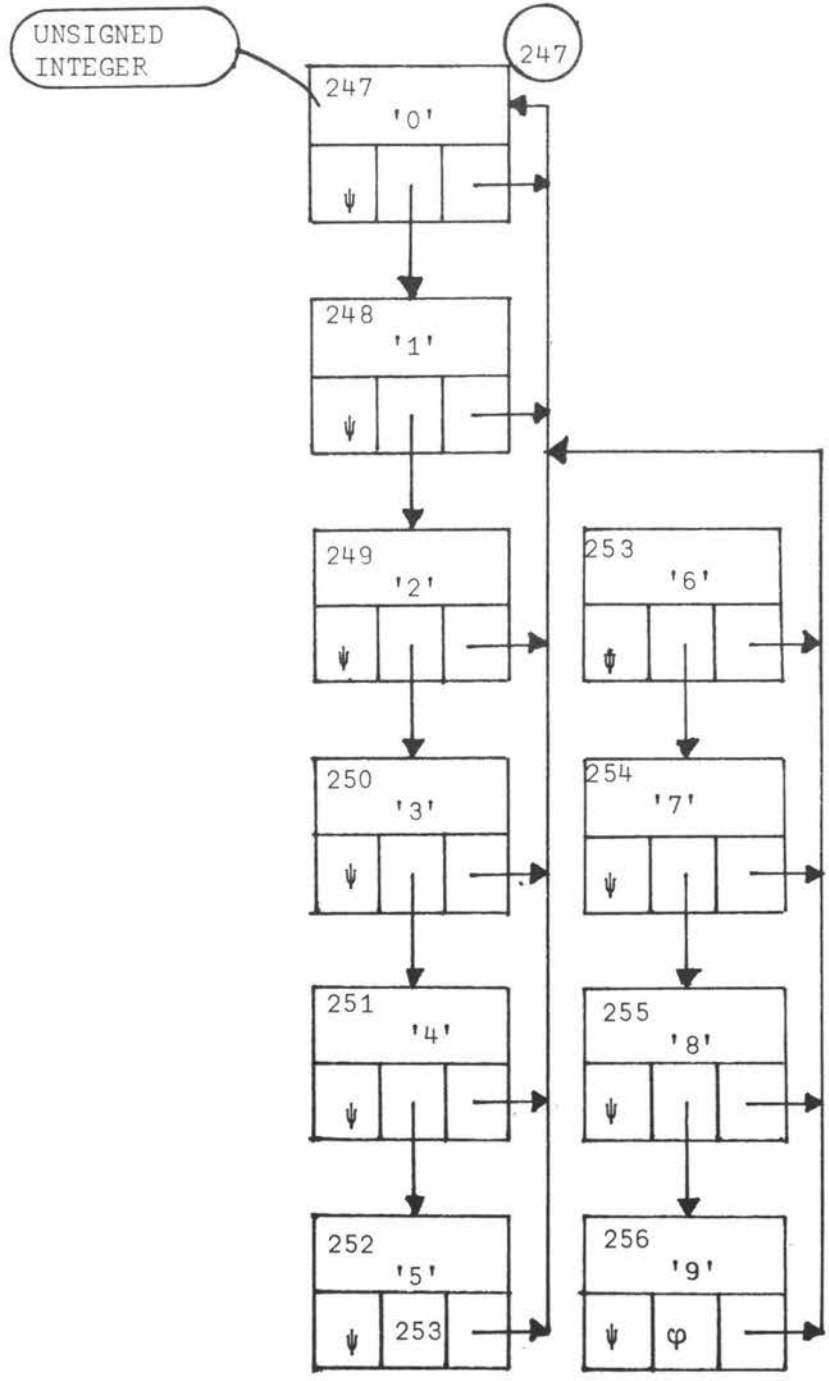

NUMBER
233
233 UNSIGNED NUMBER
239 σ
234 '+'
ψ
235 '-'
ψ φ

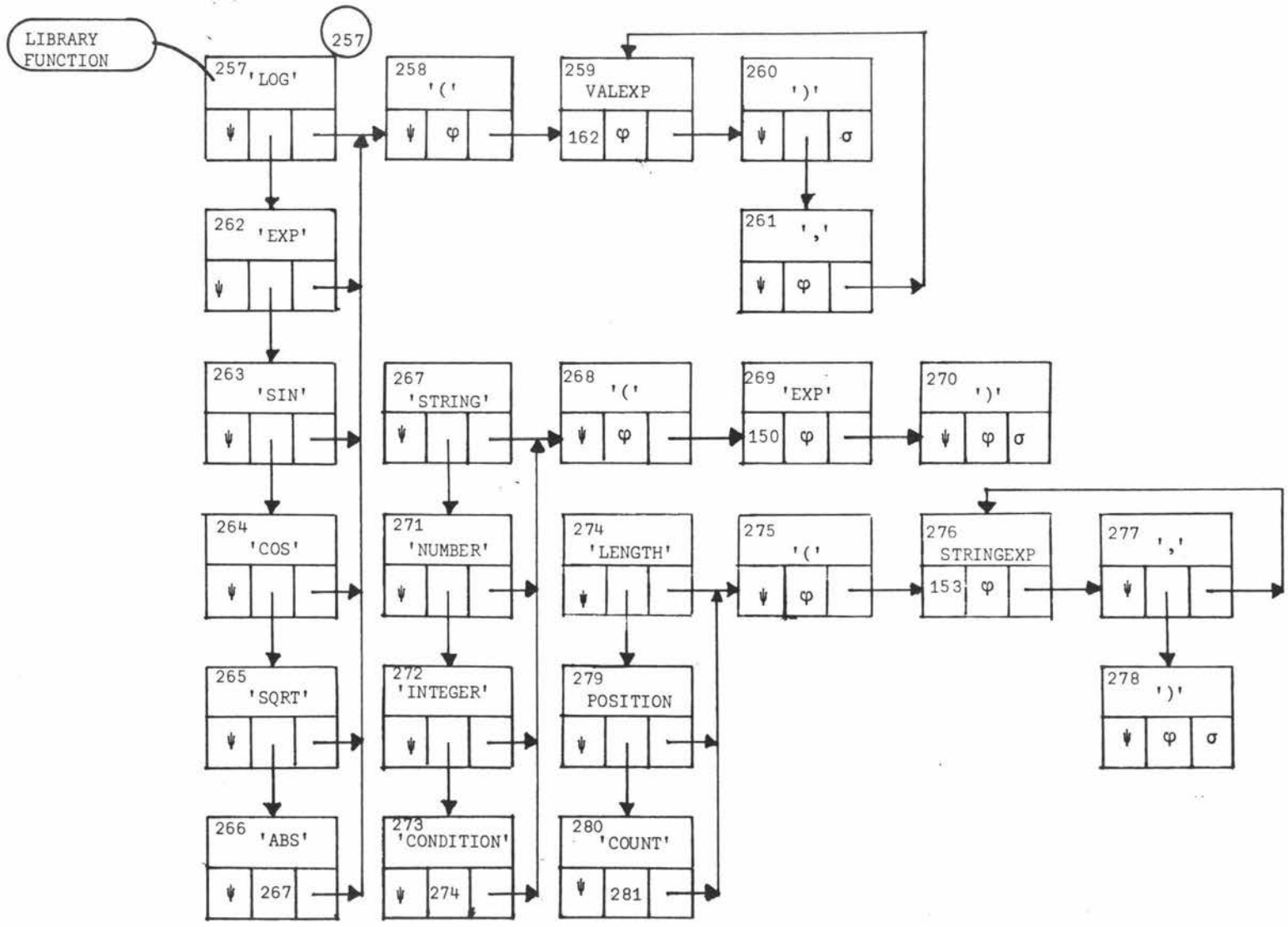

INTEGER
236
236 UNSIGNED INTEGER
247 σ
237 '+'
ψ
238 '-'
ψ φ

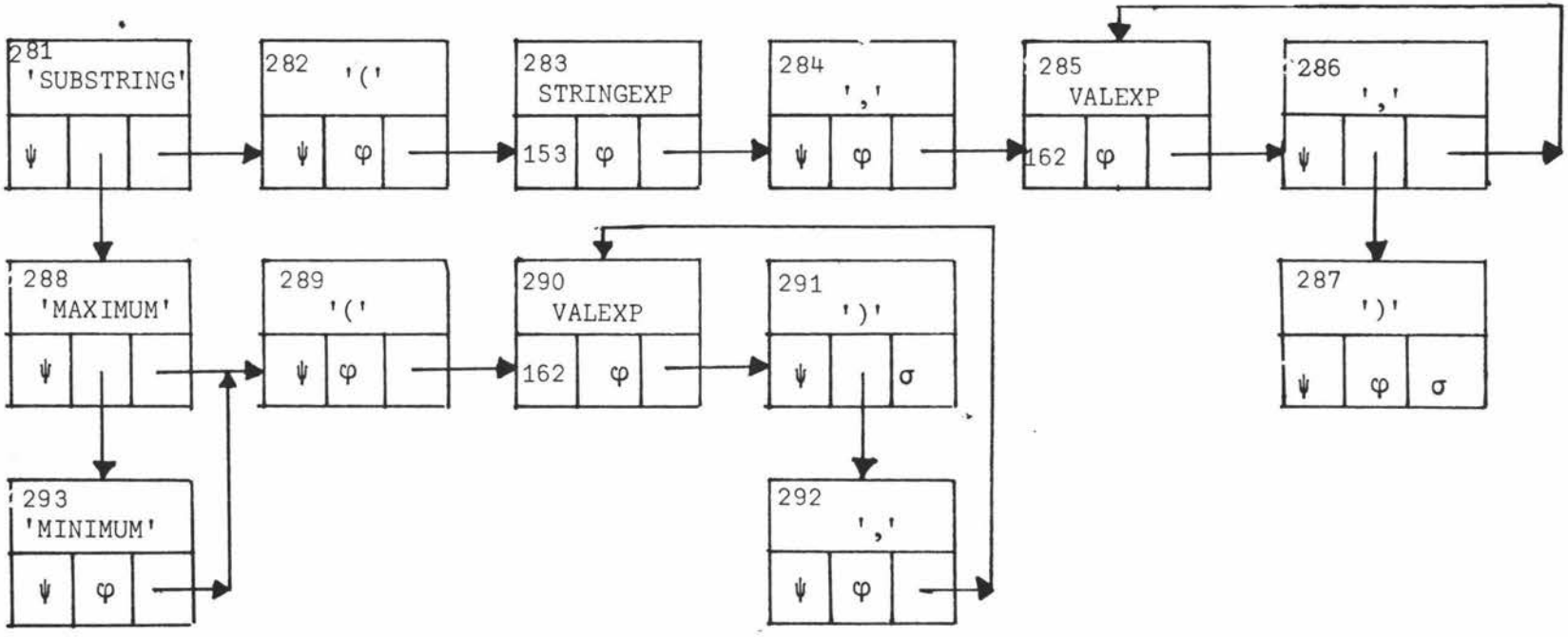

EXPONENT PART
241
241 'E'
242 INTEGER
ψ φ
236 φ σ

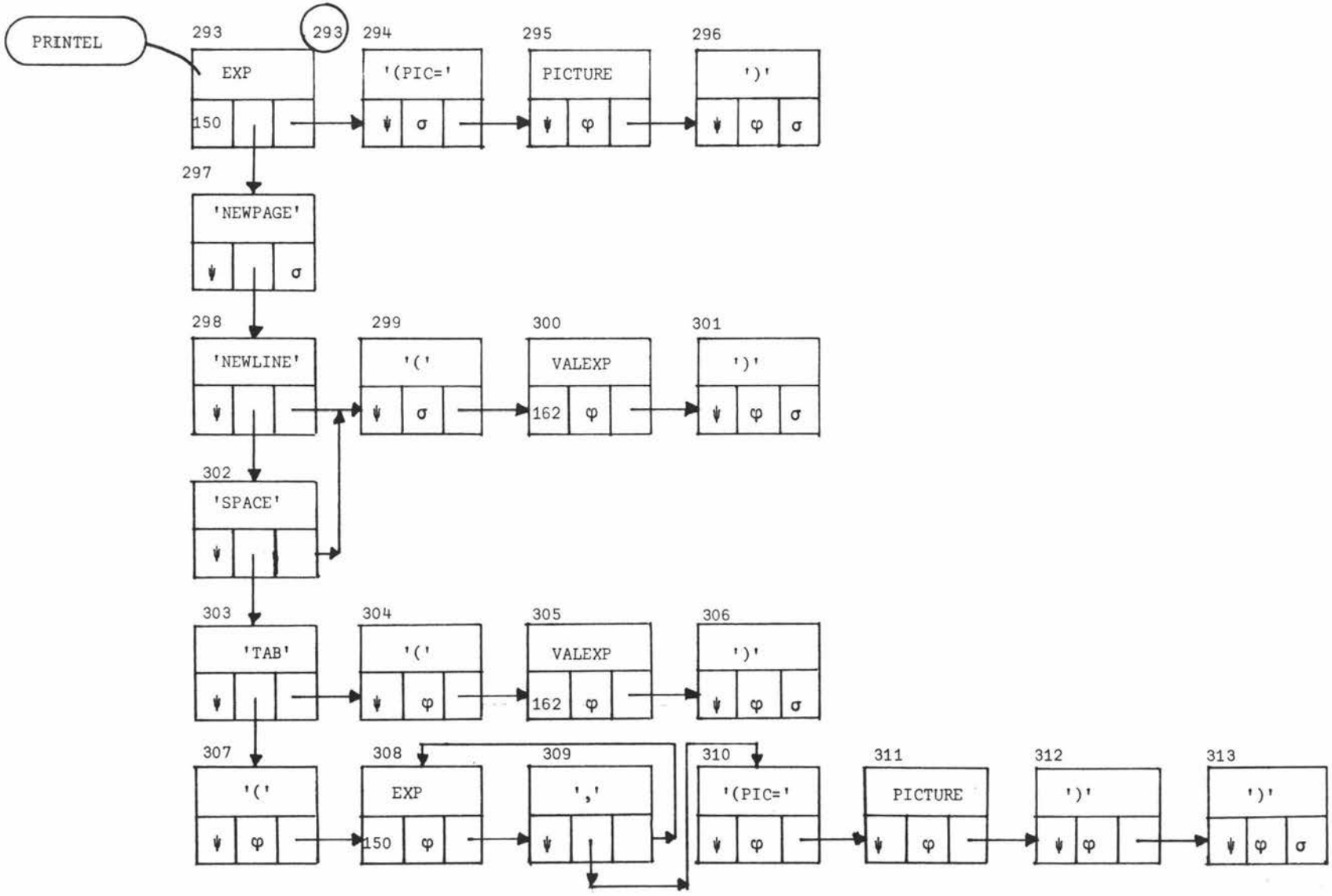

DECIMAL NUMBER
243
243 UNSIGNED INTEGER
247
244 DECIMAL FRACTION
245 φ σ

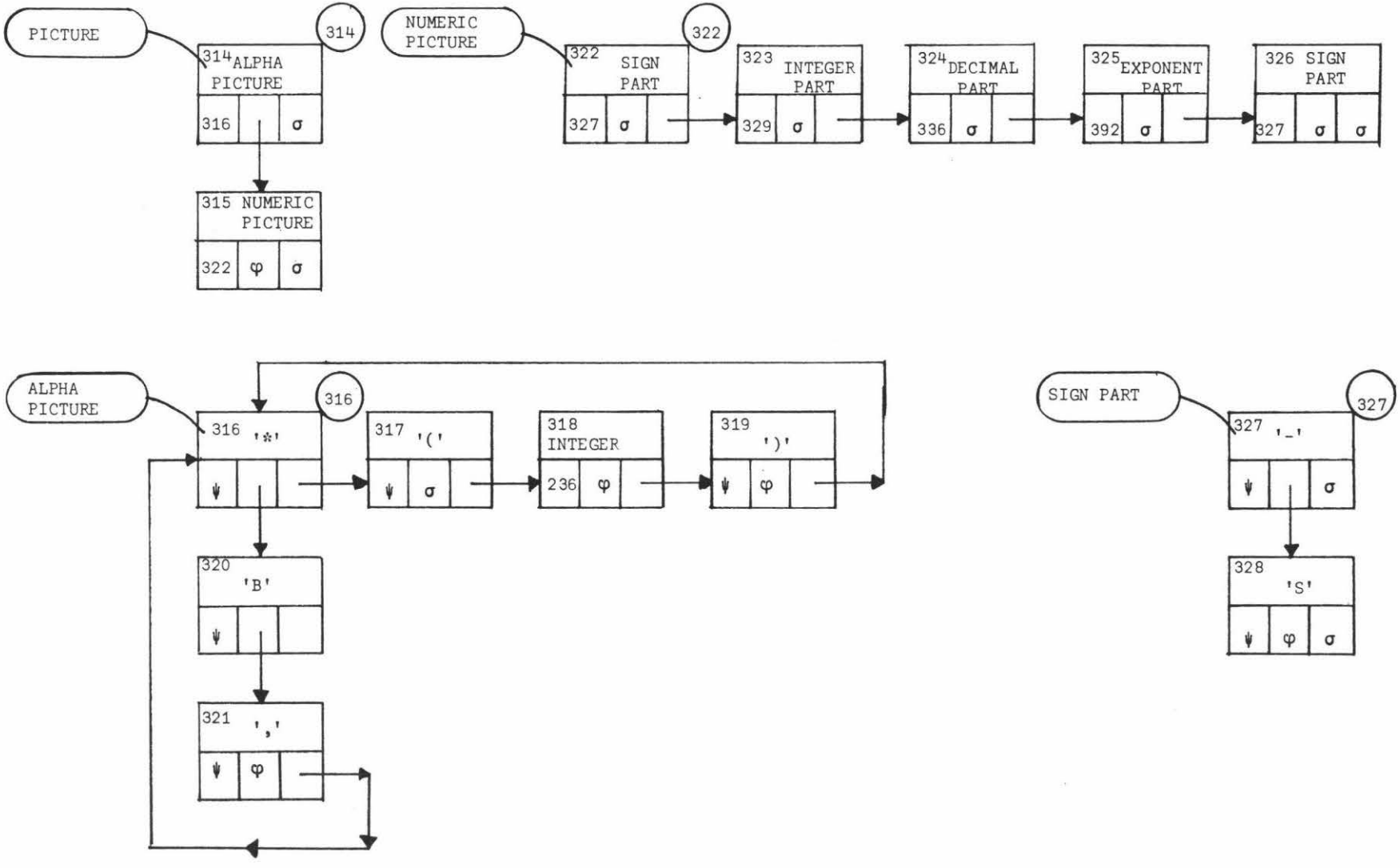

DECIMAL FRACTION
245
245 '.'
246 UNSIGNED INTEGER
ψ φ
247 φ σ

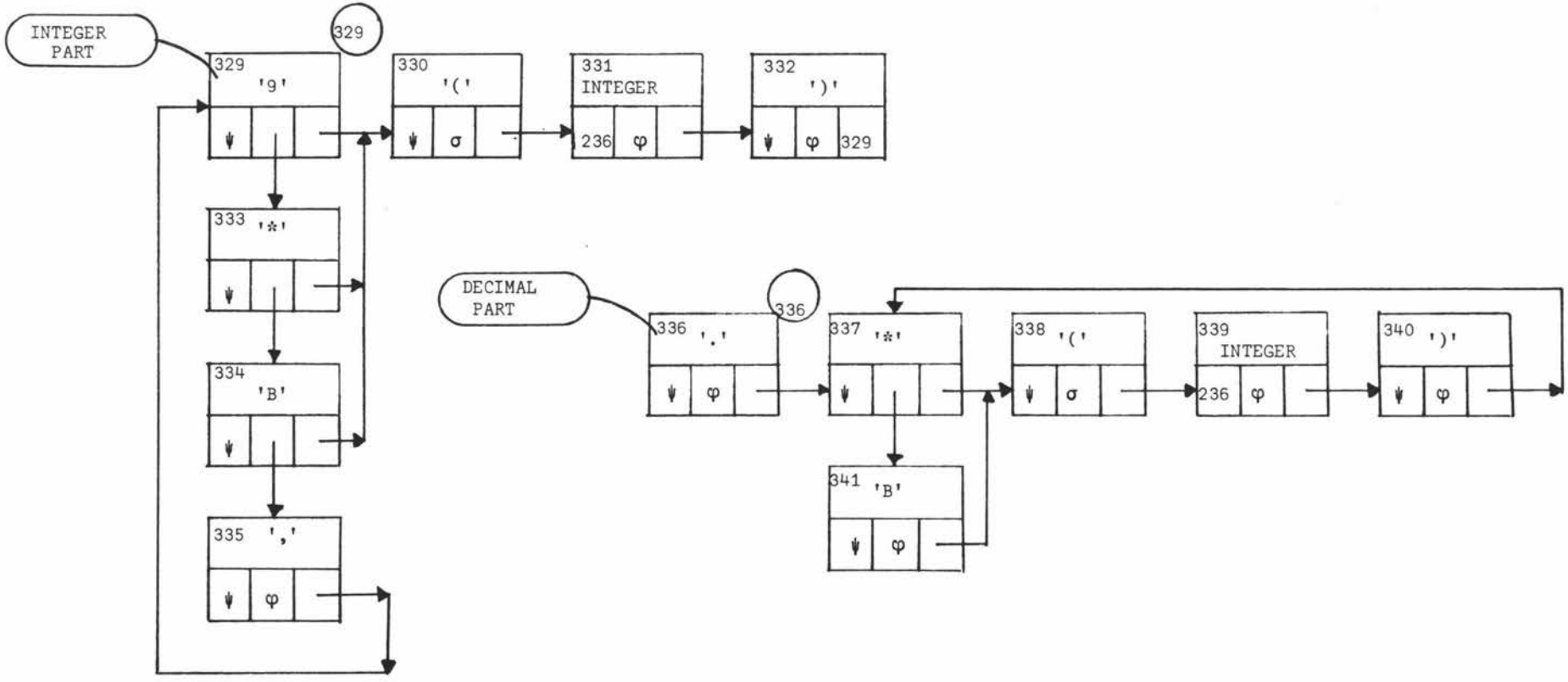

UNSIGNED INTEGER
247
247 '0'
248 '1'
249 '2'
250 '3'
251 '4'
252 '5'
253
253 '6'
254 '7'
255 '8'
256 '9'
ψ φ

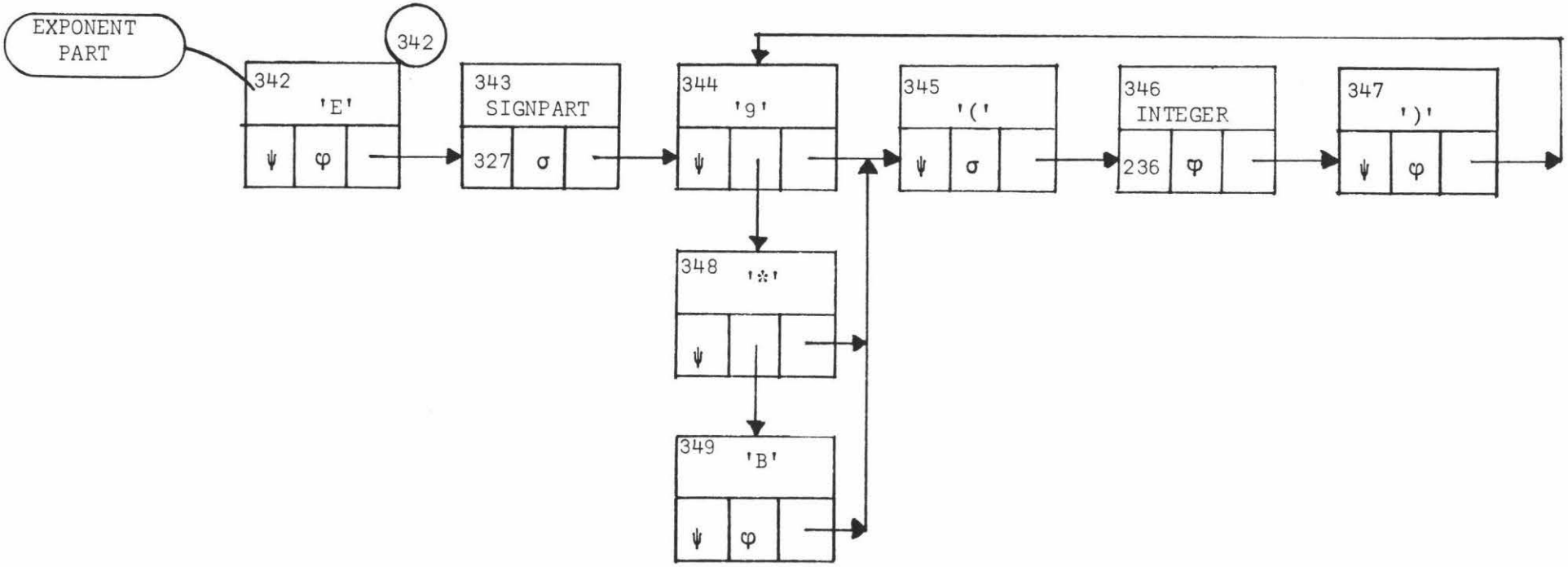

LIBRARY FUNCTION
257
257 'LOG'
258 '('
259 VALEXP
162
260 ')'
261 ','
262 'EXP'
263 'SIN'
264 'COS'
265 'SQRT'
266 'ABS'
267
267 'STRING'
268 '('
269 'EXP'
150
270 ')'
271 'NUMBER'
272 'INTEGER'
273 'CONDITION'
274
274 'LENGTH'
275 '('
276 STRINGEXP
153
277 ','
278 ')'
279 POSITION
280 'COUNT'
281


281
'SUBSTRING'
ψ
282
'('
ψ
φ
283
STRINGEXP
153
φ
284
','
ψ
φ
285
VALEXP
162
φ
286
','
ψ
287
')'
ψ
φ
σ
288
'MAXIMUM'
ψ
289
'('
ψ
φ
290
VALEXP
162
φ
291
')'
ψ
σ
293
'MINIMUM'
ψ
φ
292
','
ψ
φ


PRINTEL
293
293
EXP
150
294
'(PIC='
σ
295
PICTURE
φ
296
')'
φ
σ
297
'NEWPAGE'
σ
298
'NEWLINE'
299
'('
σ
300
VALEXP
162
φ
301
')'
302
'SPACE'
303
'TAB'
304
'('
φ
305
VALEXP
306
307
'('
308
EXP
309
','
310
'(PIC='
311
PICTURE
312
')'
313
')'


PICTURE
314
314 ALPHA PICTURE
316
σ
315 NUMERIC PICTURE
322
φ
σ
NUMERIC PICTURE
322
322 SIGN PART
327
σ
323 INTEGER PART
329
σ
324 DECIMAL PART
336
σ
325 EXPONENT PART
392
σ
326 SIGN PART
327
σ
σ
ALPHA PICTURE
316
316 '*'
317 '('
318 INTEGER
236
φ
319 ')'
320 'B'
321 ','
SIGN PART
327
327 '-'
328 'S'


INTEGER PART
329
329 '9'
330 '('
331 INTEGER
236
332 ')'
329
333 '*'
334 'B'
335 ','
DECIMAL PART
336
336 '.'
337 '*'
338 '('
339 INTEGER
236
340 ')'
341 'B'


EXPONENT PART
342
342 'E'
ψ φ
343 SIGNPART
327 σ
344 '9'
ψ
345 '('
ψ σ
346 INTEGER
236 φ
347 ')'
ψ φ
348 '*'
ψ
349 'B'
ψ φ

APPENDIX C

SUMMARY OF THE INTERMEDIATE LANGUAGE OPERATIONS

OPERATOR		INSTRUCTION FORM	
MNEMONIC	MEANING	OPCODE	PARAMETER(S)
LN	LINE NUMBER	01	***
TA	TAKE ADDRESS	02	***
TR	TAKE RESULT	03	***
TN	TAKE NUMBER	04	**********
TNO	TAKE NUMBER ZERO	05	
TN1	TAKE NUMBER ONE	06	
TB	TAKE BOOLEAN	07	**
TS	TAKE STRING	08	**...**≠
TNS	TAKE NULL STRING	09	
INDA	INDEX ADDRESS	10	
INDR	INDEX RESULT	11	
ST	STORE	12	
STA	STORE ALSO	13	
VI	VALUE IS	15	
UJ	UNCONDITIONAL JUMP	16	****
BA	BRANCH ADDRESS	17	****
IFJ	IF FALSE JUMP	18	****
OP	OPERATOR	19	**
RT	REPEAT TIMES	20	****,***
RF	REPEAT FOR	21	****,***
RFWT	REPEAT FOR WITHOUT TEST	22	***
RS	REPEAT SET	23	**,****,***
CS	CASE	24	**

cont'd

OPERATOR		INSTRUCTION FORM	
MNEMONIC	MEANING	OPCODE	PARAMETER(S)
CSWB	CASE WITHOUT BOUNDS	25	**
RAP	RETURN ADDRESS PROCEDURE	26	
RAF	RETURN ADDRESS FUNCTION	27	
CP	CALL PROCEDURE	28	***,**
CLP	CALL LIBRARY PROCEDURE	29	**,**
PE	PROCEDURE ENTRY	30	**,**
TLA	TAKE LOCAL ADDRESS	31	**
TLR	TAKE LOCAL RESULT	32	**
MSF	MAKE STORAGE FUNCTION	33	**,*,*
RE	RETURN	34	
END	END	35	
RD	READ	36	
PR	PRINT	37	
PRC	PRINT CONTROL	38	**,***.....**0≠
NP	NEW PAGE	39	
NL	NEW LINE	40	**
SP	SPACE	41	**
TAB	TAB	42	***

APPENDIX D

EXPLANATION OF INTERMEDIATE LANGUAGE OPERATIONS

Line Number (LN***)

Function of op is to update line No. register so that interpreter can keep track of position in source program and hence print meaningful error diagnostics. In general translator inserts ln op at start of object code corresponding to each line of source program. Parameter is appropriate line No. of source program.

End Operator (END)

Marks end of program.

Expressions

Translated into reverse polish form.

New Operators Used -

TR***

Parameter specifies a S.T. address. Function of op is to place the value at this address, together with its type, at the top of the main interpreter stack.

TN**********

Parameter specifies a number. Function of op is to place this number at the top of the stack, together with a 'number' type digit.

TNO,TN1

Used to place the constant 0 or 1 respectively at the top of the stack, together with a 'number' type digit.

TB*

Parameter here is a Boolean value. Function of op is to place this value at the top of the stack, together with a 'Boolean' type digit.

TS**...**≠

Parameter here is a (variable length) string. Function of op is to place this string in the string area, and place its address accompanied by a 'string address' type digit at the top of the stack.

TNS

Places representation of null string at top of stack together with a 'string' type digit.

OP**

Parameter here specifies one of the following operations -
.OR.,.AND.,.NOT.,.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.,+,-,*,/,
./.,NEG,**,CAT.
In the case of a unary op, the interpreter applies the appropriate operation to the value at the top of the stack (after first checking its type).
In the case of a binary op, the interpreter first checks the types of the top 2 values on the stack. If they are correct, it then applies the operation to these 2 values, wipes them from the stack, and stores the result and type in their place at the top of the stack.

TA***

Parameter here specifies a S.T. address. Function of op is to place this address at the top of the stack accompanies by a 'symbol table address' type digit.

INDR

Used in the case of a subscripted variable. The translator compiles object code to stack the address of the array word, and the value of each subscript expression. The op INDR follows and this uses the stacked values and array word to obtain the address of the appropriate array alement. The used stack values are then wiped from the stack and replaced by the value and type of the addressed array element.

Example

Source Code

```
-A*12+1 .GE. C(I,J*K) .AND.TRUE
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	TR	A
5	OP	'NEG'
9	TN	'12'
20	OP	'*'
24	TN1	
26	OP	'+'
30	TA	C
35	TR	I
40	TR	J
45	TR	K
50	OP	'*'
54	INDR	
56	OP	'.GE.'
60	TB	'TRUE'
63	OP	'.AND.'
67	...	

Assignment Instructions

New Operators Used -

INDA

As for INDR, except that the address of the array element, rather than its value, is placed at the top of the stack. This is accompanied by the appropriate address type.

ST

Value and type at top of stack are stored at address contained in second-to-top stack position. Top 2 stack values are then wiped.

STA

Performs storage as to ST. However stored value and type moves down one position in the stack to replace the stored into address. This position now becomes the top of the stack. (Op used in translating multiple assignments).

Example

Source Code

```
SET A,B(I,J*K),C TO D (0,4)     (LINE NO.10)
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'10'
5	TA	A
10	TA	B
15	TR	I
20	TR	J
25	TR	K
30	OP	'*'
34	INDA	
36	TA	C
41	TA	D
46	TNO	
48	TN	'4'
59	INDR	
61	STA	
63	STA	
65	ST	
67	...	

General Choice - Groups, Conditional - Groups, IF - Groups

New Operators Used -

UJ****.

Parameter is an I.L. address. I.C. is reset to this value.

IFJ****

Parameter is an I.L. address. Interpreter examines Boolean value (after checking type) at top of the stack. If false, the I.C. is reset to the value of the parameter. If true the I.C. is incremented by 6 to point to the next I.L. instruction. In either case, the top of the stack is then wiped.

Example

Source Code

```
(LINE #)
    5              DO IF B
    6                   THEN SET A TO 1
    7                   ELSE SET A TO 0
    8              END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'5'
5	TR	B
10	IFJ	(36)
16	LN	'6'
21	TA	A
26	TN1	
28	ST	
30	UJ	(50)
36	LN	'7'
41	TA	A
46	TNO	
48	ST	
50	...	

Repeated Group Instructions

Simple Repeat Without Control

Example

Source Code

```
(LINE #)
  13          DO REPEAT
  14               SET I TO I+1
  15               IF I .GT. N
  16               THEN EXIT
  17          END
```

Intermediate Language Generated

POSITION	OF	PARAMETER(S)
0	LN	'13'
5	UJ	(17)
11	UJ	(82)
17	LN	'14'
22	TA	I
27	TR	I
32	TN1	
34	OP	'+'
38	ST	
40	LN	'15'
45	TR	I
50	TR	N
55	OP	'.GT.'
59	IFJ	(76)
65	LN	'16'
70	UJ	(11)
76	UJ	(17)
82	...	

Repeat-while Instruction

Example

Source Code

```
(LINE ≠)
   4        DO REPEAT WHILE X .GT. N
   5            SET X TO X-1
   6            IF B THEN EXIT
   7        END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	UJ	(12)
6	UJ	(88)
12	LN	'4'
17	TR	X
22	TR	N
27	OP	'.GT.'
31	IFJ	(6)
37	LN	'5'
42	TA	X
47	TR	X
52	TN1	
54	OP	'-'
58	ST	
60	LN	'6'
65	TR	B
70	IFJ	(82)
76	UJ	(6)
82	UJ	(12)
88	...	

Repeat-Until Instruction

Example

Source Code

```
(LINE ≠ )
   9          DO REPEAT UNTIL X .GT. N
  10               SET X TO X-1
  11               IF B THEN EXIT
  12          END
Intermediate Language Generated
POSITION     OR      PARAMETER(S)
   0         UJ        (41)
   6         UJ        (92)
  12         LN         '9'
  17         TR         X
  22         TR         N
  27         OP       '.GT.'
  31         OP       '.NOT.'
  35         IFJ       (6)
  41         LN       '10'
  46         TA         X
  51         TR         X
  56         TN1
  58         OP       '-'
  62         ST
  64         LN       '11'
  69         TR         B
  74         IFJ      (86)
  80         UJ       (6)
  86         UJ       (12)
  92         ...
```

Repeat for 'LOC' from 'VALEXP' (to 'VALEXP') (by 'VALEXP')

New Operatores used

BA****

Parameter here specifies the address of the start of the I.L. immediately following the repeat instruction. The op is placed at the head of the object code corresponding to any REPEAT FOR and REPEAT SET instruction and causes a branch to the parameter address,as for a UJ op.It is immediately preceded by a UJ op to bypass it on entry to the loop. Inside the loop, all exits refer to this BA op.

RF****,***

Parameters as for RT op. Translator compiles object code to initially assemble the address of the controlled variable, the starting value, the test value, & the incrementing value (note - if omitted, assumed to be 1) at the top of the stack. The RF routine then stores these values, wipes them from the stack, and uses them to reset & test the controlled variable at the start of each execution of the loop. The first parameter is ued to branch out on completion of the loop - the second is used to reset the line No. register at the start of each execution of the loop.

Example

Source Code

```
(LINE #
   5          DO REPEAT FOR A FROM 1 TO N BY 2
   6               SET B(A) TO A*A
   7          END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	UJ	(12)
6	BA	(88)
12	LN	'5'
17	TA	A
22	TN1	
24	TR	N
29	TN	'2'
40	RF	(6,5)
49	LN	'6'
54	TA	B
59	TR	A
64	INDA	
66	TR	A
71	TR	A
76	OP	'*'
80	ST	
82	UJ	(40)
88	...	

Repeat for 'LOC' from 'VALEXP' (by 'VALEXP')

New Operator used

RFWT***

Similar to the RF op. However source program instruction provides no test value - hence the RFWT routine requires no branch out address.

Example

Source Code

```
(LINE ≠)
   8        DO REPEAT FOR A FROM 1 BY 2
   9             SET B(A) TO A*A
  10             IF C THEN EXIT
  11        END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	UJ	(12)
6	BA	(101)
12	LN	'8'
17	TA	A
22	TN1	
24	TN	'2'
35	RFWT	'8'
40	LN	'9'
45	TA	B
50	TR	A
55	INDA	
57	TR	A
62	TR	A
67	OP	'*'
71	ST	
73	LN	'10'
78	TR	C
83	IFJ	(95)
89	UJ	(6)
95	UJ	(35)
101	...	

REPEAT FOR 'LOC' SET TO 'EXP','EXP','EXP'...

New Operator used.

RS**

Parameter here specifies the No. of set elements.

Example

Source Code

```
(LINE ≠)
   19          DO REPEAT FOR A SET TO 1,2,3
   20               SET B(A) TO A*A
   21          END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	UJ	(12)
6	BA	(123)
12	LN	'19'
17	TA	A
22	TN1	
24	STSA	
26	UJ	(74)
32	LN	'19'
37	TN	'2'
48	STSA	
50	UJ	(74)
56	LN	'19'
61	TN	'3'
72	ST	
74	LN	'20'
79	TA	B
84	TR	A
89	INDA	
91	TR	A
96	TR	A
101	OP	'*'
105	ST	
107	RS	'3'
111	UJ	(56)
117	UJ	(32)
123	...	

Note - The RS routine controls which following UJ instruction is taken at each execution of the loop. The parameter specifying the No. of set elts enables the routine to determine when the loop has been completed, at which point it must reset its internal state variables.

Repeat-times Instruction

New Operator used

RT****,***

The translator compiles object code to initially assemble the value of the times expression at the top of the stack. This is then stored by the RT routine, wiped from the stack, and used by the routine to control the No. of times the loop is executed. The op's first parameter is an I.L. address pointing to the BA op for the loop - this enables the routine to branch out on completion of the loop. The second parameter is the source program line No. of the start of the repeat instruction - this enables the RT routine to update the line No. register at the start of each execution of the loop.

Example

Source Code

```
(LINE #)
   0          DO REPEAT N TIMES
   1              SET X TO X+1
   2          END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	UJ	(12)
6	UJ	(60)
12	LN	'0'
17	TR	N
22	RT	(6,0)
31	LN	'1'
36	TA	X
41	TR	X
46	TN1	
48	OP	'+'
52	ST	
54	UJ	(22)
60	...	

CASE - Group Instruction With Bounds

New Operator used

CS**

Parameter here is the No. of case instructions within the CASE - Group. The Translator compiles object code to assemble the values of the CASE expression and bounds at the top of the Stack. The CS routine then uses these values to check and determine which case statement to branch to.

Example

Source Code

```
3          DØ CASE J IN (2,4) OF
4               SET X TØ 2
5               SET X TØ 3
6               SET X TØ 4
7          END
```

Intermediate Language Generated

POSITIØN	OP	PARAMETER(S)
0	LN	'3'
5	TR	J
10	TN	'2'
21	TN	'4'
32	UJ	(125)
38	LN	'4'
43	TA	X
48	TN	'2'
59	ST	
61	UJ	(147)
67	LN	'5'
72	TA	X
77	TN	'3'
88	ST	
90	UJ	(147)
96	LN	'6'
101	TA	X
106	TN	'4'
117	ST	

119	UJ	(147)
125	CS	'3'
129	UJ	(38)
135	UJ	(67)
141	UJ	(96)
147		

CASE - Group Instruction Without Bounds

New Operator used

CSWB**

Parameter here is the No. of case instructions within the CASE - group Translation is similar to the CASE - group with bounds, a simplification here,of course,is that a lower bound of 1 is assumed. The value of the CASE expression is used to determine which CASE instruction is to be executed.

Example

Source Code

```
3     DØ CASE K + 1 OF
4          SET L TØ L - 1
5          SET L TØ L - 2
6          SET L TØ L - 3
7     END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'3'
5	TR	K
10	TN1	
12	ØP	'+'
16	UJ	(127)
22	LN	'4'
27	TA	L
32	TR	L
37	TN1	
39	OP	'-'
43	ST	
45	UJ	(149)
51	LN	'5'

```
 56          TA          L
 61          TR          L
 66          TN          '2'
 77          ØP          '-'
 81          ST
 83          UJ          (149)
 89          LN          '6'
 94          TA          L
 99          TR          L
104          TN          '3'
115          OP          '-'
119          ST
121          UJ          (149)
127          CSWB        '3'
131          UJ          (22)
137          UJ          (51)
143          UJ          (89)
149        -----         ---
```

READ - Instruction

New Operator used

RD

MUSSEL has a simple unformatted stream input, data being separated on cards by a blank or a comma. Strings have the additional delimiter '!' Arrays may be read in row-wise by the specification of only the array name. The generation of Intermediate Language for a READ instruction is a particularly easy task for the Translator. For each variable in a READ list, code is produced to place the variable's address at the top of the stack followed by a RD operation

Example

Source Code

```
 8      READ A,B,C
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'8'
5	TA	A
10	RD	
12	TA	B
17	RD	
19	TA	C
24	RD	
26	...	...

PRINT - Instruction With No PICTURE Specification

New Operator Used

PR

This is similar to the RD operator, in a READ - Instruction. Here the Translator compiles Intermediate Language to place the address, or value if a expression or constant at the top of the stack for each PRINT List element. Each such piece of code is followed by a PR operator.

Example

Source Code

```
10   PRINT X,Y (I,K), B + C * Z
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'10'
5	TA	X
10	PR	
12	TA	Y
17	TR	I
22	TR	K
27	INDR	
29	PR	
31	TR	B
36	TR	C
41	TR	Z
46	OP	'*'
50	OP	'+'

54	PR
56	...

PRINT - Instruction With PICTURE Specification

New Operator used

PRC ** ----**0≠

This operator has 2 parameters. The first parameter is the number of list elements appearing in the PIC specification. The second parameter is the PICTURE itself in alphametic form, followed by an alphameric record mark. In translating the elements involved in a simple, or multiple, PICTURE specification in MUSSEL, the Translator compiles Intermediate Language to assemble the addresses, or values of the concerned PRINT List elements at the top of the Stack. This is then followed by the PRC (Print Control) operator.

Example

Source Code

```
  10   PRINT (X,Y(I,K), B+C*Z)(PIC=**.*)
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'10'
5	TA	X
10	TA	Y
15	TR	I
20	TR	J
25	INDR	
27	TR	B
32	TR	C
37	TR	Z
42	OP	'*'
46	QP	'+'
50	PRC	(3,**,*)
64	...	

PRINT - Instruction With Printer Controls

New Operators used

NP

The appearance of NEWPAGE within a PRINT - Instruction generates a NP operation within the Intermediate Language.

NL **

The appearance of NEWLINE within a PRINT - Instruction generates a NL operation with the number of newlines to be skipped as a parameter. If no parameter specified then 1 is assumed.

SP **

The appearance of SPACE with or without a parameter, within a PRINT - Instruction generates a SP operation in the Intermediate Language with the number of spaces required as a parameter. If no parameter specified in the Source Language then Translator assumes 1.

TAB ***

The appearance of TAB with a parameter, within a PRINT - Instruction generates a TAB operation in the Intermediate Language followed by a 3 digit parameter specifying the number of positions to be tabulated.

Example

Source Code

```
 10   PRINT NEWPAGE, A, NEWLINE,B,
      SPACE(5),C, TAB(60),D.
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'10'
5	NP	
7	TA	A
12	PR	
14	NL	'1'
18	TA	B
23	PR	
25	SP	'5'
29	TA	C
34	PR	

```
36          TAB         '60'
41           TA          D
46           PR
48          ...
```

Case-Group Instruction

New operator used

CS**

Parameter here is the No. of case statements. Translator compiles object code to assemble the values of the case expression and bounds at the top of the stack. The CS routine then uses these values to check and determine which case statement to branch to. (Similar to RS routine).

Example

Source Code

(LINE ≠)

```
   4          DO CASE 4 IN (3,5) OF
   5               SET A TO 1
   6               SET A TO 2
   7               SET A TO 3
   8          END
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	LN	'4'
5	TN	'4'
16	TN	'3'
27	TN	'5'
38	UJ	(122)
44	LN	'5'
49	TA	A
54	TN1	
56	ST	
58	UJ	(144)
64	LN	'6'
69	TA	A
74	TN	'2'
85	ST	
87	UJ	(144)
93	LN	'7'
98	TA	A
103	TN	'3'

cont'd

```
114         ST
116         UJ        (144)
122         CS          '3'
126         UJ         (44)
132         UJ         (64)
138         UJ         (93)
144         ...
```

Function and procedure declarations

New Operators used

PE**,**

Inserted at head of object code co-responding to procedure (or function) declaration. First parameter is the No. of formal parameters - second is the No. of local identifiers. These enable interpreter to reserve space for them in the stack.

MSF**,*,*

Used in the case of local arrays. First parameter is the local address of the last array word - second is the No. of array identifiers in the segment - third is the dimension of each array. Task of op MSF is to set up a storage mapping function, and an array word for each array in an array segment, and to allocate space in the stack for the elts of the array. In processing the reservation of a local array, the translator converts each array segment into a set of object program representations of the subscript bound expressions, followed by MSF. The MSF routine then uses these values and replaces them at the top of the stack by the storage mapping function and space for the array ELTS.

Example

Source Code

```
RESERVE (A,B,C(1:N,1:M))
```

Intermediate Language Generated

POSITION	OP	PARAMETER(S)
0	TN1	
2	TR	N
7	TN1	
9	TR	M
14	MSF	(5,3,2)
20	...	

Note - It is assumed that the array words A,B,C have been allocated local addresses 3,4,5 respectively.

TLA**

Parameter here is a local address - that is, a relative stack position above the return address for the procedure. Function of op is to first check the type of this addressed location. If an address, then the contents (type & value) of the location are placed at the top of the stack, otherwise, the local address is converted to an absolute stack address, and placed at the top of the stack, together with a 'stack address' type digit.

TLR**

Parameter here is a local address. Interpreter places addressed result and type (note - may be indirectly addressed) at the top of the stack.

VI

Used in function declaration for storing function value. Interpreter takes value and type at top of stack and stores them in the reserved stack position immediately below the return address for the function. The top of the stack is then wiped.

RE

Used when branching back from a procedure. Interpreter refers to the return address for the procedure and hence resets the I.C. all values above and including the return address are then wiped from the stack.

Function and Procedure Calls

New Operators used

RAP

Used in translation of procedure call. Reserves space in the stack for a return address.

RAF

Used in translation of function call. Reserves space in the stack for the function value and a return address (in that order).

CP***,**

First parameter here is a symbol table address. This contains a pointer to the name of procedure, and also a pointer to the start of the object code corresponding to the procedure body. The second parameter is the No. of actual parameters. In translating a procedure or function call, the translator compiles a RAP or RAF op respectively, followed by object code to assemble the addresses (or values if proper expressions) of the actual parameters at the top of the stack. The CP op then follows. This fills in the return address, sets the I.C. to the start of the object code corresponding to the procedure body, and finally stores the No. of actual parameters to be checked against the No. of formal parameters by the PE op.

Example

Source Code

```
(LINE #)
   3          DEFINE MEAN ON A,B AS
   4          DO
   5              VALUE IS (A+B)/2
   6          END
   .
   .
   .
  12          SET M TO MEAN(U,V**2)
```

Intermediate Language Generated At Function Definition

POSITION	OP	PARAMETER(S)
0	LN	'3'
5	PE	(2,0)
11	LN	'5'
16	TLR	'1'
20	TLR	'2'
24	OP	'+'
28	TN	'2'
39	OP	'/'
43	VI	
45	RE	
47	...	

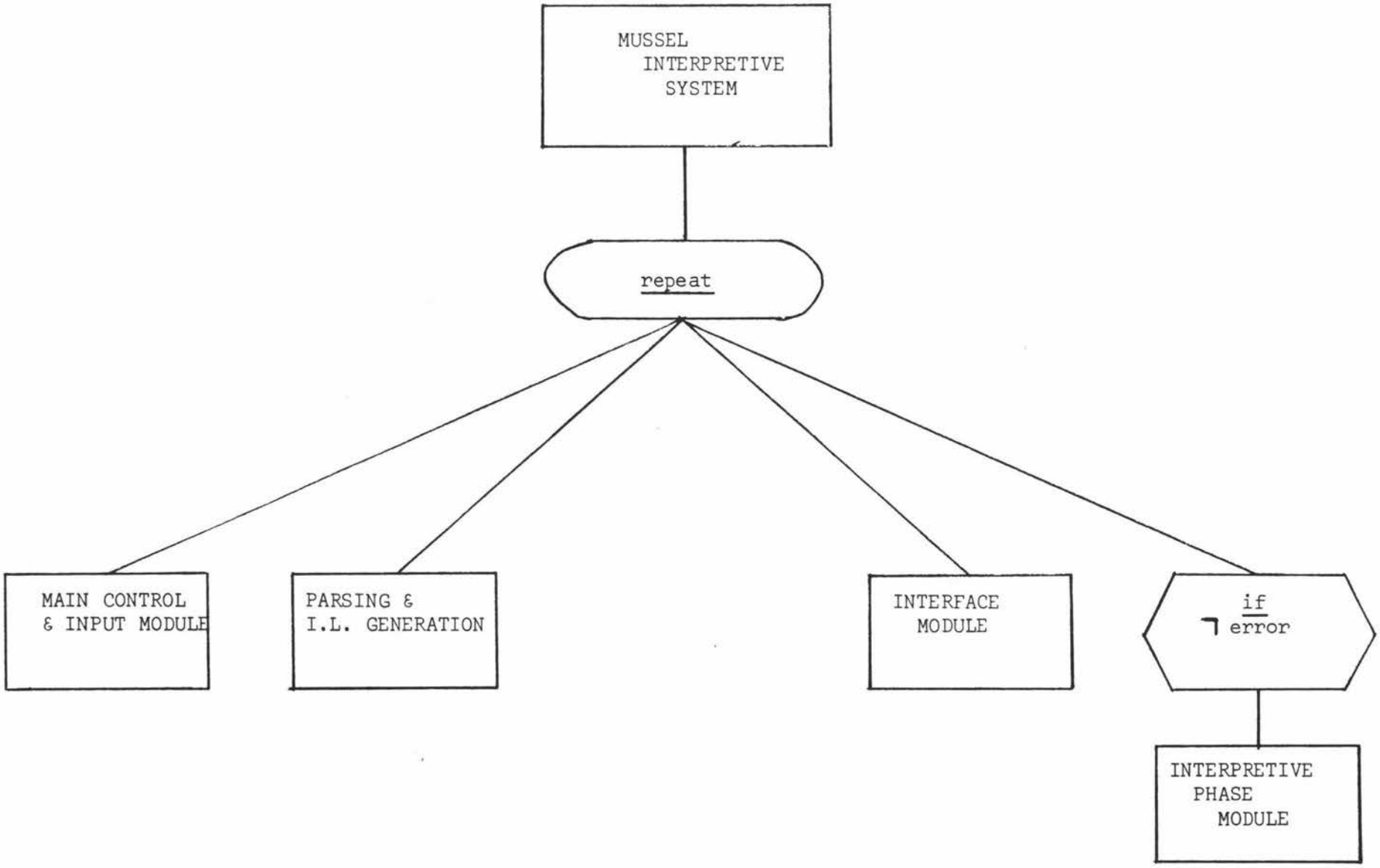
Intermediate Language Generated At Function Call.

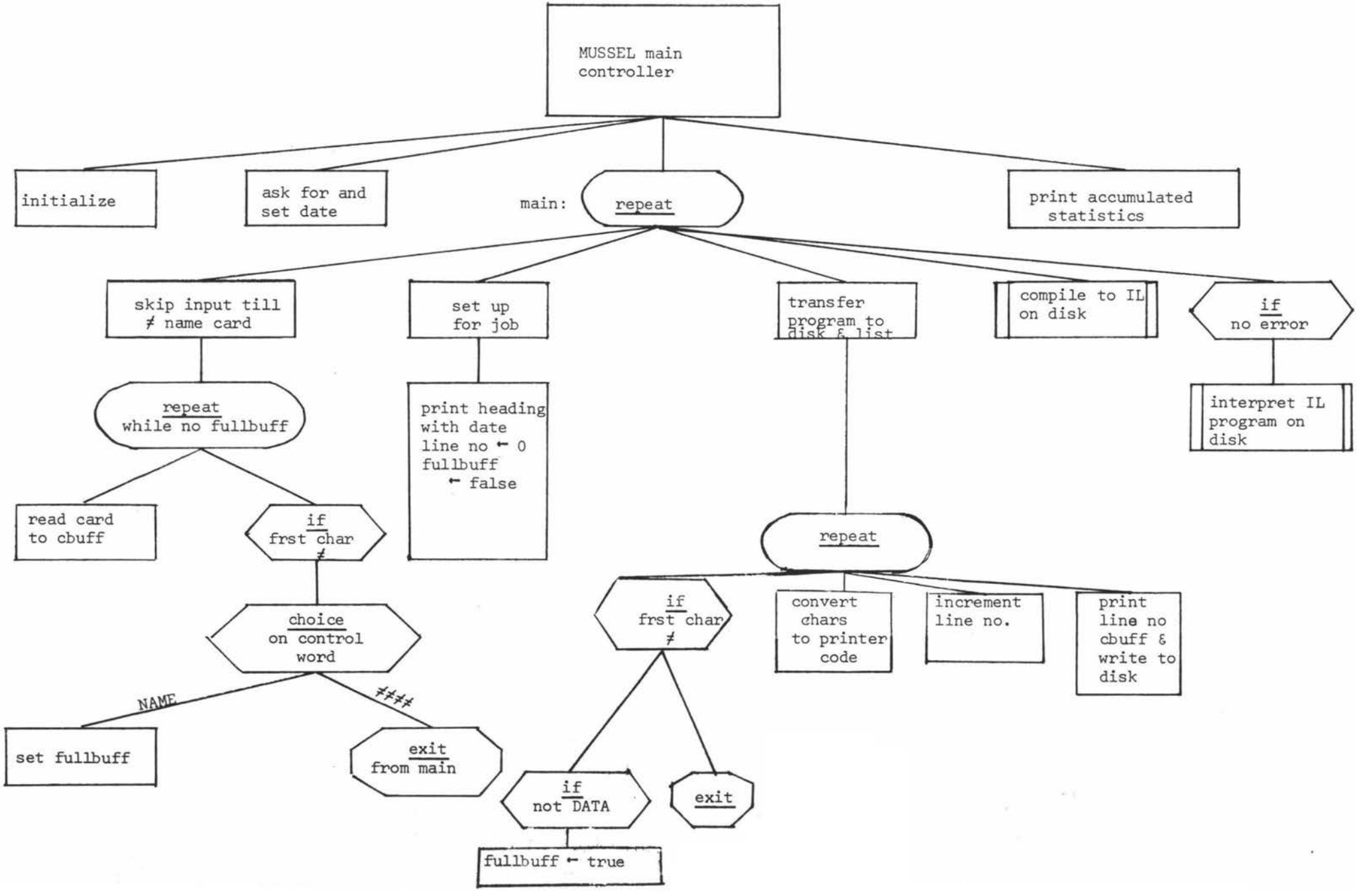
POSITION	OP	PARAMETER(S)
0	LN	'12'
5	TA	M
10	RAF	
12	TA	U
17	TR	V
22	TN	Z
22	TN	'2'
33	OP	'**'
37	CP	(MEAN,2)
44	...	

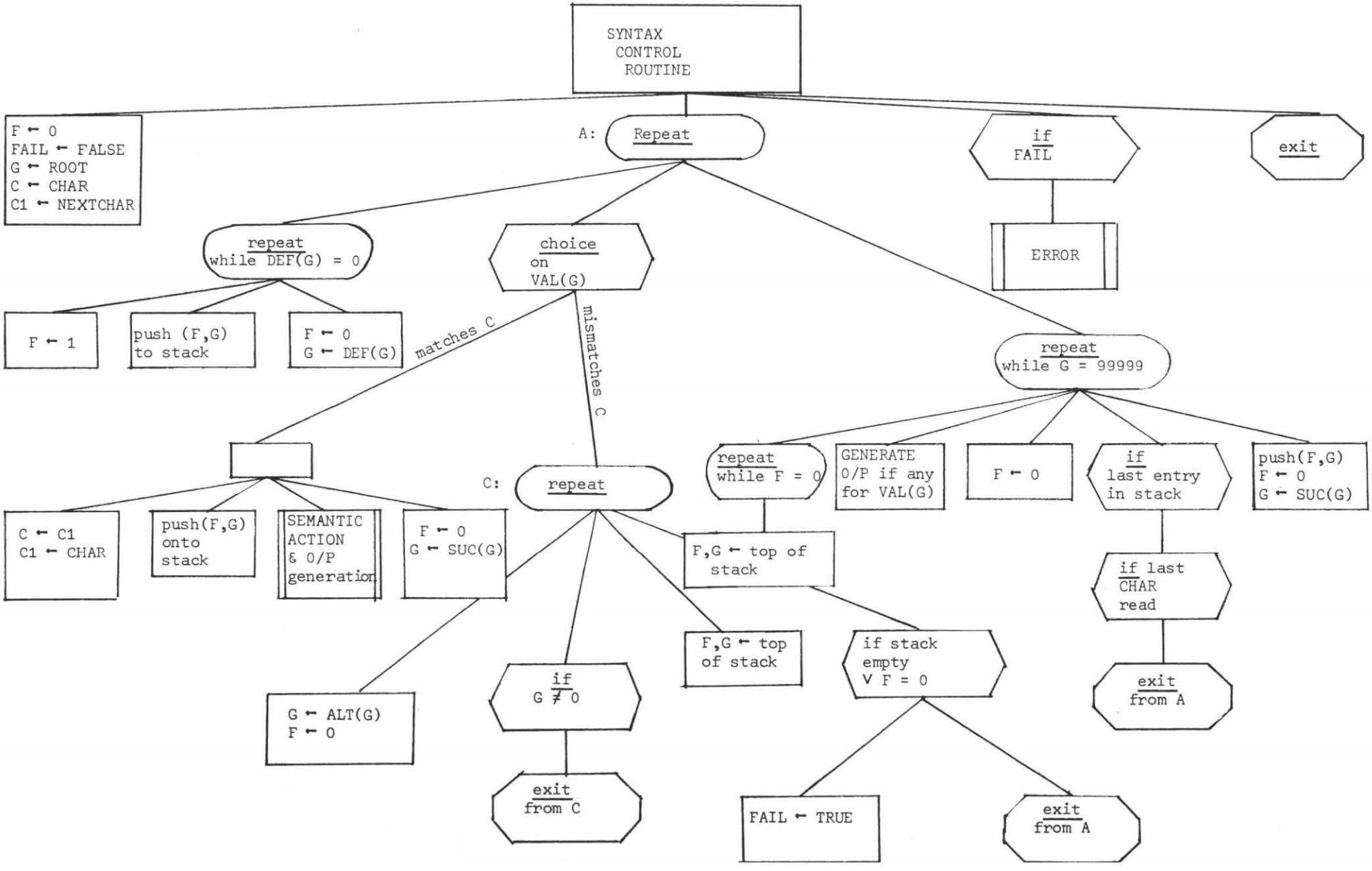
A P P E N D I X E

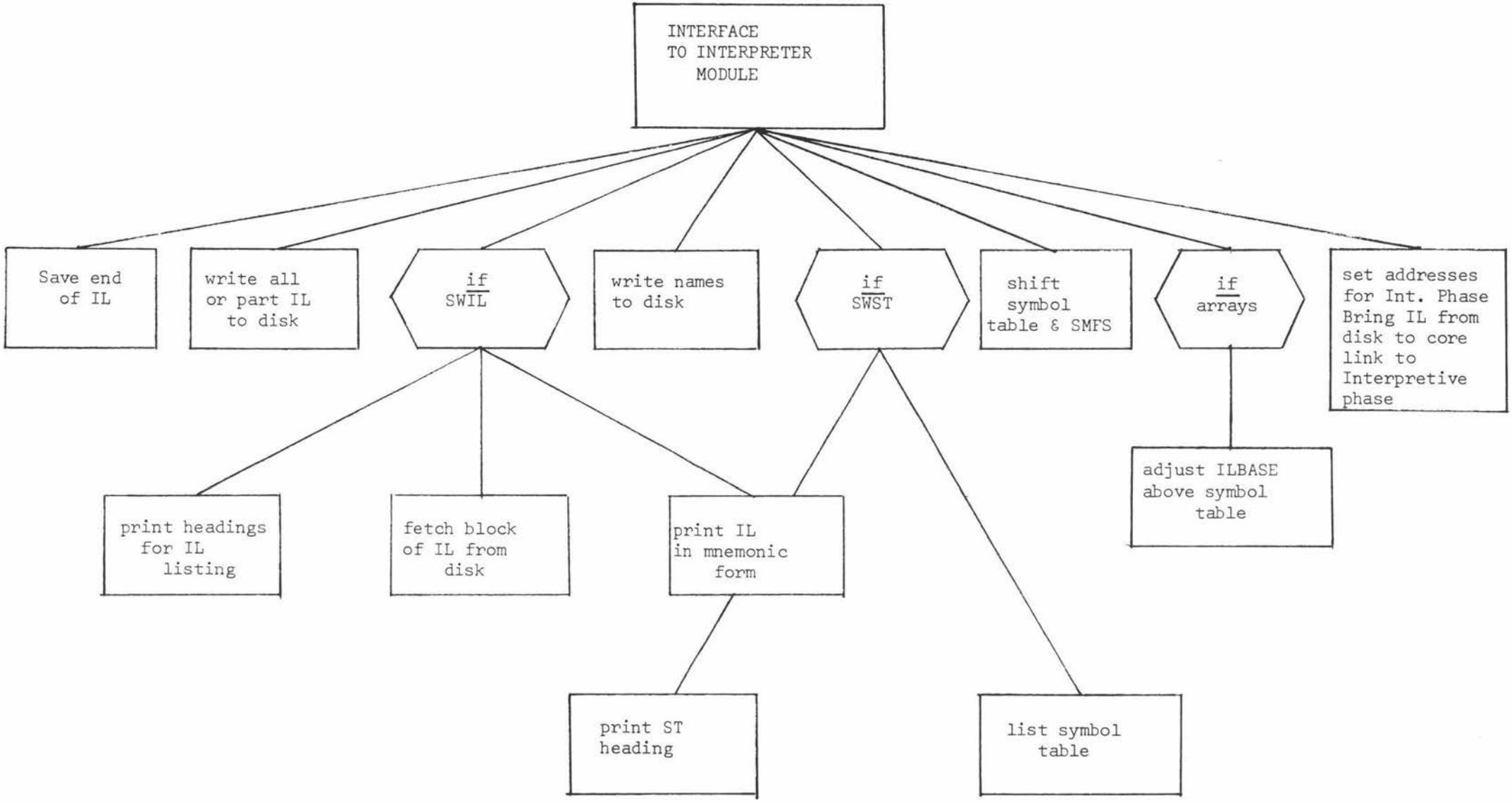
GENERAL STRUCTURE DIAGRAMS

OF THE COMPILATION PHASE


MUSSEL
INTERPRETIVE
SYSTEM
repeat
MAIN CONTROL
& INPUT MODULE
PARSING &
I.L. GENERATION
INTERFACE
MODULE
if
¬ error
INTERPRETIVE
PHASE
MODULE


MUSSEL main controller
initialize
ask for and set date
main:
repeat
print accumulated statistics
skip input till ≠ name card
set up for job
transfer program to disk & list
compile to IL on disk
if no error
repeat while no fullbuff
print heading with date line no ← 0 fullbuff ← false
interpret IL program on disk
read card to cbuff
if frst char ≠
repeat
choice on control word
NAME
++++
if frst char ≠
convert chars to printer code
increment line no.
print line no cbuff & write to disk
set fullbuff
exit from main
if not DATA
exit
fullbuff ← true


SYNTAX
CONTROL
ROUTINE
F ← 0
FAIL ← FALSE
G ← ROOT
C ← CHAR
C1 ← NEXTCHAR
A:
Repeat
if
FAIL
exit
ERROR
repeat
while DEF(G) = 0
F ← 1
push (F,G)
to stack
F ← 0
G ← DEF(G)
choice
on
VAL(G)
matches C
mismatches C
repeat
while G = 99999
C:
repeat
C ← C1
C1 ← CHAR
push(F,G)
onto
stack
SEMANTIC
ACTION
& O/P
generation
F ← 0
G ← SUC(G)
repeat
while F = 0
GENERATE
O/P if any
for VAL(G)
F ← 0
if
last entry
in stack
push(F,G)
F ← 0
G ← SUC(G)
F,G ← top of
stack
if last
CHAR
read
exit
from A
G ← ALT(G)
F ← 0
if
G ≠ 0
F,G ← top
of stack
if stack
empty
∨ F = 0
exit
from C
FAIL ← TRUE
exit
from A


INTERFACE
TO INTERPRETER
MODULE
Save end
of IL
write all
or part IL
to disk
if
SWIL
write names
to disk
if
SWST
shift
symbol
table & SMFS
if
arrays
set addresses
for Int. Phase
Bring IL from
disk to core
link to
Interpretive
phase
print headings
for IL
listing
fetch block
of IL from
disk
print IL
in mnemonic
form
adjust ILBASE
above symbol
table
print ST
heading
list symbol
table

A P P E N D I X F

EXAMPLES OF MUSSEL PROGRAMS

SOURCE LISTING:NAME BROWN,CHARLES K.

```
LINE                                STATEMENT

        #NAME    BROWN,CHARLES K.
        $CODE
        $XREF
        *
        *    PROGRAM TO PROCESS STREAM OF TELEGRAMS
        *
  1     DO
  2          RESERVE WORDCOUNT,OVERLENGTH,WORD
  3          PRINT NEWPAGE,TAB(30),!TELEGRAM ANALYSIS!,
        X           NEWLINE,TAB(30),!**********!,NEWLINE(3)
  4     OUTA:DO REPEAT
  5               SET WORDCOUNT TO 0
  6               SET OVERLENGTH TO FALSE
  7               DO REPEAT
  8                    READ WORD
  9                    DO CHOICE OF
 10                         IF WORD .EQ. !! THEN EXIT FROM OUTA
 11                         IF WORD .EQ. !ZZZZ!
 12                         THEN DO
 13                                   PRINT NEWLINE,!WORDCOUNT = !,WORDCOUNT(PIC=***)
 14                                   IF OVERLENGTH
 15                                   THEN PRINT SPACE(5),!**CHECK**!
 16                                   EXIT
 17                              END
 18                         IF WORD .NE. !STOP!
 19                         THEN DO
 20                                   SET WORDCOUNT TO WORDCOUNT+1
 21                                   IF LENGTH(WORD) .GT. 12
 22                                   THEN SET OVERLENGTH TO TRUE
 23                              END
```

;OURCE LISTING:NAME BROWN,CHARLES K.

```
LINE                                STATEMENT

 24                         END
 25                         PRINT WORD,SPACE
 26                    END
 27               END OUTA
 28               PRINT NEWLINE(3),!**END OF JOB**!
 29          END
```

COMPILER LISTING:NAME BROWN,CHARLES K.

```
LINE                                STATEMENT

          #NAME    BROWN,CHARLES K.
          *
          *     PROGRAM TO PROCESS STREAM OF TELEGRAMS
          *
    1     DO
    2          RESERVE WORDCOUNT,OVERLENGTH,WORD
    3          PRINT NEWPAGE,TAB(30),!TELEGRAM ANALYSIS!,
          X           NEWLINE,TAB(30),!**********!,NEWLINE(3)
    4     OUTA:DO REPEAT
    5               SET WORDCOUNT TO 0
    6               SET OVERLENGTH TO FALSE
    7               DO REPEAT
    8                    READ WORD
    9                    DO CHOICE OF
   10                         IF WORD .EQ. !! THEN EXIT FROM OUTA
   11                         IF WORD .EQ. !ZZZZ!
   12                         THEN DO
   13                                   PRINT NEWLINE,!WORDCOUNT = !,WORDCOUNT(PIC=***)
   14                                   IF OVERLENGTH
   15                                   THEN PRINT SPACE(5),!**CHECK**!
   16                                   EXIT
   17                              END
   18                         IF WORD .NE. !STOP!
   19                         THEN DO
   20                                   SET WORDCOUNT TO WORDCOUNT+1
   21                                   IF LENGTH(WORD) .GT. 12
   22                                   THEN SET OVERLENGTH TO TRUE
   23                              END
   24                    END
   25                    PRINT WORD,SPACE
```

COMPILER LISTING:NAME BROWN,CHARLES K.

```
    LINE                                STATEMENT

      26                  END
      27              END OUTA
      28              PRINT NEWLINE(3),!**END OF JOB**!
      29          END

COMPILATION COMPLETE   0 ERROR(S) DETECTED
  29 STATEMENTS PROCESSED
 NO INDENT ERROR(S)
```

INTERMEDIATE LANGUAGE LISTING:NAME BROWN,CHARLES K.

POSITION	MNEMONIC	M/C CODE	PARAMETER(S)
0000	LN	01	003
0005	NP	39	
0007	TAB	42	030
0012	TS	08	!TELEGRAM ANALYSIS!
0050	PRC	38	01,
0056	NL	40	01
0060	TAB	42	030
0065	TS	08	!**********!
0103	PRC	38	01,
0109	NL	40	03
0113	LN	01	004
0118	UJ	16	0130
0124	UJ	16	0546
0130	LN	01	005
0135	TA	02	'WORDCOUNT'
0140	TNO	05	
0142	ST	12	
0144	LN	01	006
0149	TA	02	'OVERLENGTH'
0154	TB	07	00
0158	ST	12	
0160	LN	01	007
0165	UJ	16	0177
0171	UJ	16	0531
0177	LN	01	008
0182	TA	02	'WORD'
0187	RD	36	
0189	LN	01	009
0194	UJ	16	0207
0200	UJ	16	0515

:NTERMEDIATE LANGUAGE LISTING:NAME BROWN,CHARLES K.

)OSITION	MNEMONIC	M/C CODE	PARAMETER(S)
0206	LN	01	010
0211	TR	03	'WORD'
0216	TNS	09	
0218	OP	19	'EQ'
0222	IFJ	18	0235
0228	UJ	16	0546
0234	LN	01	011
0239	TR	03	'WORD'
0244	TS	08	!ZZZZ!
0256	OP	19	'EQ'
0260	IFJ	18	0391
0266	LN	01	013
0271	NL	40	01
0275	TS	08	!WORDCOUNT = !
0303	PRC	38	01,
0309	TR	03	'WORDCOUNT'
0314	PRC	38	01,***
0326	LN	01	014
0331	TR	03	'OVERLENGTH'
0336	IFJ	18	0380
0342	LN	01	015
0347	SP	41	05
0351	TS	08	!**CHECK**!
0373	PRC	38	01,
0379	LN	01	016
0384	UJ	16	0201
0390	LN	01	018
0395	TR	03	'WORD'
0400	TS	08	!STOP!
0412	OP	19	'NE'

INTERMEDIATE LANGUAGE LISTING:NAME BROWN, CHARLES K.

POSITION	MNEMONIC	M/C CODE	PARAMETER(S)
0416	IFJ	18	0504
0422	LN	01	020
0427	TA	02	'WORDCOUNT'
0432	TR	03	'WORDCOUNT'
0437	TN1	06	
0439	OP	19	'+'
0443	ST	12	
0445	LN	01	021
0450	TA	02	'WORD'
0455	CLP	29	CO,01
0461	TN	04	00
0472	OP	19	'GT'
0476	IFJ	18	0499
0482	LN	01	022
0487	TA	02	'OVERLENGTH'
0492	TB	07	01
0496	ST	12	
0498	LN	01	023
0503	LN	01	024
0508	UJ	16	0201
0514	LN	01	025
0519	TR	03	'WORD'
0524	PR	37	
0526	SP	41	01
0530	LN	01	026
0535	UJ	16	0171
0541	LN	01	027
0546	UJ	16	0124
0552	LN	01	028
0557	NL	40	03

INTERMEDIATE LANGUAGE LISTING:NAME BROWN,CHARLES K.

POSITION	MNEMONIC	M/C CODE	PARAMETER(S)
0561	TS	08	!**END OF JOB**!
0593	PRC	38	01,
0599	LN	01	029
0604	END	35	

SYMBOL TABLE LISTING:NAME BROWN,CHARLES K.

INDEX	NAME PTR	OFLO	TYPE	VALUE/PTR(S)	NAME
017	9981	000	0	000000000	'OVERLENGTH'
095	9953	000	J	0546	'OUTA'
096	9961	000	0	000000000	'WORD'
117	9999	000	0	000000000	'WORDCOUNT'

SOURCE LISTING:NAME SMITH,AGNES

```
LINE                                STATEMENT

          #NAME     SMITH,AGNES
          *
          *     READ SEX AND OCCUPATION CODES AND PRINT DESCRIPTIONS
          *
     1    DO
     2         RESERVE SEX,OCCUPATION,
     3         READ SEX,OCCUPATION
     4         DO CHOICE OF
     5              IF SEX.EQ.0 THEN PRINT NEWLINE !MALE!
     6              IF SEX.EQ.1 THEN PRINT NEWLINE !FEMALE!
     7         END
     8         DO CHOICE OF
     9              IF OCCUPATION .EQ.0 THEN PRINT TAB(10) !BUTCHER!
    10              IF OCCUPATION .EQ.1 THEN PRINT TAB(10) !BAKER!
    11              IF OCCUPATION .EQ.2 THEN PRINT TAB(10) !CANDLESTICKMAKER!
    12         END
    13    END
```

COMPILER LISTING:NAME SMITH,AGNES

```
    LINE                                STATEMENT

          #NAME     SMITH,AGNES
          *
          *     READ SEX AND OCCUPATION CODES AND PRINT DESCRIPTIONS
          *
      1   DO
      2         RESERVE SEX,OCCUPATION,
      3         READ SEX,OCCUPATION
      4         DO CHOICE OF
      5               IF SEX.EQ.0 THEN PRINT NEWLINE !MALE!
      6               IF SEX.EQ.1 THEN PRINT NEWLINE !FEMALE!
      7         END
      8         DO CHOICE OF
      9               IF OCCUPATION .EQ.0 THEN PRINT TAB(10) !BUTCHER!
     10               IF OCCUPATION .EQ.1 THEN PRINT TAB(10) !BAKER!
     11               IF OCCUPATION .EQ.2 THEN PRINT TAB(10) !CANDLESTICKMAKER!
     12         END
     13   END
```

COMPILATION COMPLETE 0 ERROR(S) DETECTED
13 STATEMENTS PROCESSED
NO INDENT ERROR(S)

SOURCE LISTING:NAME GORDON,HAMISH

```
LINE                                STATEMENT

        #NAME     GORDON,HAMISH
        *
        * PROGRAM TO READ A NAME(FORENAMES AND SURNAME)
        *  PRINT SURNAME FOLLOWED BY INITIALS WITH FULL STOP AFTER EACH INITIAL
        *
  1     DO
  2          RESERVE NAME,INITIALS,SURNAME,I
  3          READ NAME
  4          SET INITIALS TO !!
  5               DO REPEAT
  6                    SET I TO POSITION(! !,NAME)
  7                    DO IF I.EQ.0 THEN
  8                         DO
  9                              SET SURNAME TO NAME
 10                              EXIT
 11                         END
 12                         ELSE DO
 13                                   SET INITIALS TO SUBSTRING(NAME,1,1).CAT.!.!.CAT.
        X                                 INITIALS
 14                                   SET NAME TO SUBSTRING(NAME,I+1,LENGTH(NAME))
 15                              END
 16                         END
 17                   END
 18         PRINT SURNAME,INITIALS
 19     END
```

COMPILER LISTING:NAME GORDON,HAMISH

```
    LINE                                STATEMENT

          #NAME     GORDON,HAMISH
          *
          * PROGRAM TO READ A NAME(FORENAMES AND SURNAME)
          *  PRINT SURNAME FOLLOWED BY INITIALS WITH FULL STOP AFTER EACH INITIAL
          *
      1   DO
      2        RESERVE NAME,INITIALS,SURNAME,I
      3        READ NAME
      4        SET INITIALS TO !!
      5             DO REPEAT
      6                  SET I TO POSITION(! !,NAME)
      7                  DO IF I.EQ.0 THEN
      8                       DO
      9                            SET SURNAME TO NAME
     10                            EXIT
     11                       END
     12                       ELSE DO
     13                                 SET INITIALS TO SUBSTRING(NAME,1,1).CAT.!.!.CAT.
          X                                  INITIALS
     14                                 SET NAME TO SUBSTRING(NAME,I+1,LENGTH(NAME))
     15                            END
     16                       END
     17                  END
     18        PRINT SURNAME,INITIALS
     19   END

COMPILATION COMPLETE   0 ERROR(S) DETECTED
  19 STATEMENTS PROCESSED
 NO INDENT ERROR(S)
```

SOURCE LISTING:NAME BRUSH,BASIL

```
LINE                                STATEMENT

          #NAME     BRUSH,BASIL
          *
          *     READ 100 NUMBERS AND PRINT IN REVERSE ORDER 10/LINE
          *
    1     DO
    2          RESERVE I,A(1:100)
    3          READ A
    4          DO REPEAT FOR I FROM 100 TO 1 BY - 1
    5          TEST:DO IF I./.10*I .EQ. I
    6               THEN DO
    7                         PRINT NEWLINE,A(I)(PIC=-*(10).*(4))
    8                         EXIT FROM TEST
    9                    END
   10               ELSE DO
   11                         PRINT A(I)(PIC=-*(10).*(4))
   12                         EXIT FROM TEST
   13                    END
   14               END TEST
   15          END
   16     END
```

COMPILER LISTING:NAME BRUSH,BASIL

```
     LINE                                STATEMENT

               ±NAME     BRUSH,BASIL
               *
               *    READ 100 NUMBERS AND PRINT IN REVERSE ORDER 10/LINE
               *
       1       DO
       2            RESERVE I,A(1:100)
       3            READ A
       4            DO REPEAT FOR I FROM 100 TO 1 BY - 1
       5            TEST:DO IF I./.10*I .EQ. I
       6                 THEN DO
       7                           PRINT NEWLINE,A(I)(PIC=-*(10).*(4))
       8                           EXIT FROM TEST
       9                      END
      10                 ELSE DO
      11                           PRINT A(I)(PIC=-*(10).*(4))
      12                           EXIT FROM TEST
      13                      END
      14                 END TEST
      15            END
      16       END
```

COMPILATION COMPLETE 0 ERROR(S) DETECTED
16 STATEMENTS PROCESSED
NO INDENT ERROR(S)