THE IMPLEMENTATION OF FORMULA ALGOL IN FSL

by

Renato Iturriaga, Tim Standish, Rudy Krutar,
and Jay Earley

Carnegie Institute of Technology
Pittsburgh, Pennsylvania
October 12, 1966

This work was supported by the Advanced Research Projects
Agency of the Office of the Secretary of Defense (SD-146).

ABSTRACT

This paper describes how FSL was used to implement Formula Algol as it existed in October, 1965. Some changes have been made in Formula Algol since that date, and, consequently, this paper does not give an exact description of the current running system. Nevertheless, it reveals various classes of compiler mechanisms and techniques for using FSL that should be of value to anyone desiring to understand how FSL is used to implement complex compilers of the Algol family. It also gives insight into compilation techniques that can be used to implement formula manipulation and list processing.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF CHARTS, GRAPHS, AND ILLUSTRATIONS

ERRATA FOR

THE IMPLEMENTATION OF FORMULA ALGOL IN FSL

Note: A prerequisite for reading "The Implementation of Formula Algol in FSL" is to have read Jerome A* Feldman's doctoral dissertation entitled, "A Formal Semantics for Computer Languages".

PAGE	LOCATION	CORRECTION
3	1st line of production subroutine	add label 'CHG' to first production
3	3rd line of production subroutine	delete '+' from end of word 'TYPE'
4	line 18	add ',' after words 'identifier list'
6	line 9	insert ')' after 'i_m-lowerbound$_n$'
7	6th and 7th lines from bottom	switch CLA i to read LXP L,R0 LXP L,R0 CLA i
7	5th line from bottom	should be 'TRA V48'
8	line 6	delete ','
9	line 15	replace ',' at end of line with word 'is'
10	10th line from bottom	change '←←' to '↵'
11	11th line	insert words 'is produced' after $\overset{*}{\text{B}\quad\text{C}}$
12	17th line	insert word 'position' after RIGHT2
15	line 10	should be 'STD T1'
15	line 14	should be 'ADD 0 1'
15	insert after line 21	'ADD 0 3'
15	line 27	delete commas
16	line 4	replace 'cell' with 'cells'
18	line 16	change 'run-' to 'compile-'
18	line 19	delete commas surrounding 'therefore'
18	line 25	delete commas surrounding 'therefore'
19	line 6	insert 'the' between 'of' and 'code'
19	line 11	delete comma after 'label'
19	2nd line from bottom	change 'T ← LAB[LEFT2 . . . . $];' to read 'T ← LAB[LEFT2 , , , , $];'

PAGE	LOCATION	CORRECTION
20	line 8	delete commas surrounding 'therefore'
20	line 14	add hyphen to 'code' at end of line
22	7th line from bottom	change 'EXEC 45' to read 'EXEC 15'
23	line 6	change 'EXEC 35' to read 'EXEC 15'
23	line 12	change 'mark transfer to a routine X35' to 'transfer to a switching routine V48 indirectly through X35'
23	lines 19 and 20	underline '<u>for</u>'
24	line 4	underline '<u>for</u>'
25	3rd line from bottom	underline '<u>for</u>'
25	last line	underline '<u>for</u>'
26	line 13	insert 'CODELOC' before arrow '→'
26	line 15	change 'e' to 'E'
26	3rd line from bottom	replace 'T ← 4' with 'T ← E5'
27	line 1	replace 'a - E' with 'an E - '
27	line 7	insert ', in turn, ' after 'This'
27	line 16	insert before 'EXEC 26' 'Except for additions needed to handle recursion, which are discussed in the sequel,'
29	line 2	replace 'β CLA β' with 'β CLA B'
29	line 5	insert '0' between 'LXP' and 'VCP'
30	line 12	insert 'the' after 'in'
31	line 19	change 'identifier' to identifiers'
32	line 17	delete '-' in 'LENGTHOF(-CRADLE)'
32	7th line from bottom	replace 'CALL(I)' with 'HEAD(I)'
33	directly beneath page number 33	add parenthesis to line 'identifier with tagged with class'

PAGE	LOCATION	CORRECTION
33	lower left hand corner	add 'TRM' beneath word '(parameter)'
33	lower right hand corner	add arrow from box $\boxed{A \leftarrow B}$ down to bottom line
34	lines 7, 8 and 9	sentence beginning in line 7 is incomplete and is repeated in complete form beginning in line 9. Remove incomplete sentence.
35	line 17	delete 'x' from 'TYPE PROCEDURE x | '
35	line 19	delete 'X' from 'SECX' to get 'SEC'
36	line 4	remove '(' before '(FPL' to get 'FPL'
37	line 1	change 'contest' to 'context'
37	line 23	change SOnow' to ' (So now'
38	line 8	change 'page 39' to 'page 29'
38	line 11	change 'pages 4 and 5' to 'page 4'
38	2nd line from bottom	change 'A 2 FALSE' to 'A 2 TRUE'
39	line 3	change 'exec' to 'EXEC'
40	line 2	delete '(' after 'MARKJUMP[DECLARE];'
40	line 6	change 'see page 40' to 'see page 30'
40	line 9	put ',' between α and 2 in '005α, 2'
41	line 7	change ',' to ';'
41	6th line from bottom	insert ';' after 'CODE(JUMP[V202])' and change 'p46' to 'p35'
41	5th line from bottom	change 'pp 44-45' to 'pp 33-34'
42	line 9	change 'this' to 'This'
42	line 10	change 'page 46' to 'page 35'
42	line 16	change 'p53' to 'p40'

PAGE	LOCATION	CORRECTION
46	line 12	change 'assign L3-Storloc' to 'assign L3 = Storloc'
48	line 12	fix split infinitive
49	line 6	change 'X' to 'X7'
49	line 21	change 'x7' to 'X7' and change 'CSS>' to '<CSS>'
49	line 16	change 'here' to 'Here' and put '.' at end of sentence
49	line 25	change 'here' to 'Here'
49	line 35	change 'here' to 'Here'
49	last line	change 'POP[LSS[CSS]' to 'POP[LSS,CSS]'
50	line 26	delete phrase 'with different dotted lines
52	line 16	insert 'P(Y+Z)' after 'call statement'
52	line 17	delete 'P(Y+Z)'
56	line 13	add 'e' to 'relativ'
56	line 6	delete 'a'
60	3rd box down left hand side	should have YES attached to entrance and should read ↓ YES $\boxed{\text{Compute V55(B)}}$
62	flowchart	arrowheads missing
63	line 4	replace 'Exponents' with 'Exponentiate'
64	second diagram	put a '-' over last box in diagram
65	line 19	change 'PURPLE' to 'RED'
68	line 2	change first ',' in list to '.'
69	line 13	change ',then' to '.Then' starting a new sentence
69	line 18	delete commas around ',second,'
74	line 16	put '⌒' marks in 'A B C' getting A⌒B⌒C'

THE FLOW OF SYSTEMS

Three separate operations are needed to produce the Formula Algol compiler. First, the productions defining the syntax of the language are processed by means of a GATE program called the production loader. The output of this program is a set of syntax tables which are stored on tape for later use. Second, the formal semantic routines defining the semantics of the language are processed in the FSL system producing, as output, a set of semantic tables. These tables are also stored on tape for later use. Third, and finally, a system called MAGIC reads in the syntax tables and the semantic tables, and by use of these tables operates as a compiler for source language statements. The source language statements are read in by MAGIC and translated into an object program. The object program is then run provided no errors have been detected during compilation. During the initialization of the object program a collection of run-time routines is read into the memory. These run-time routines constitute a set of well-defined actions that are executed upon call by the object program. Figure 1 on page two shows this flow of systems diagramatically.

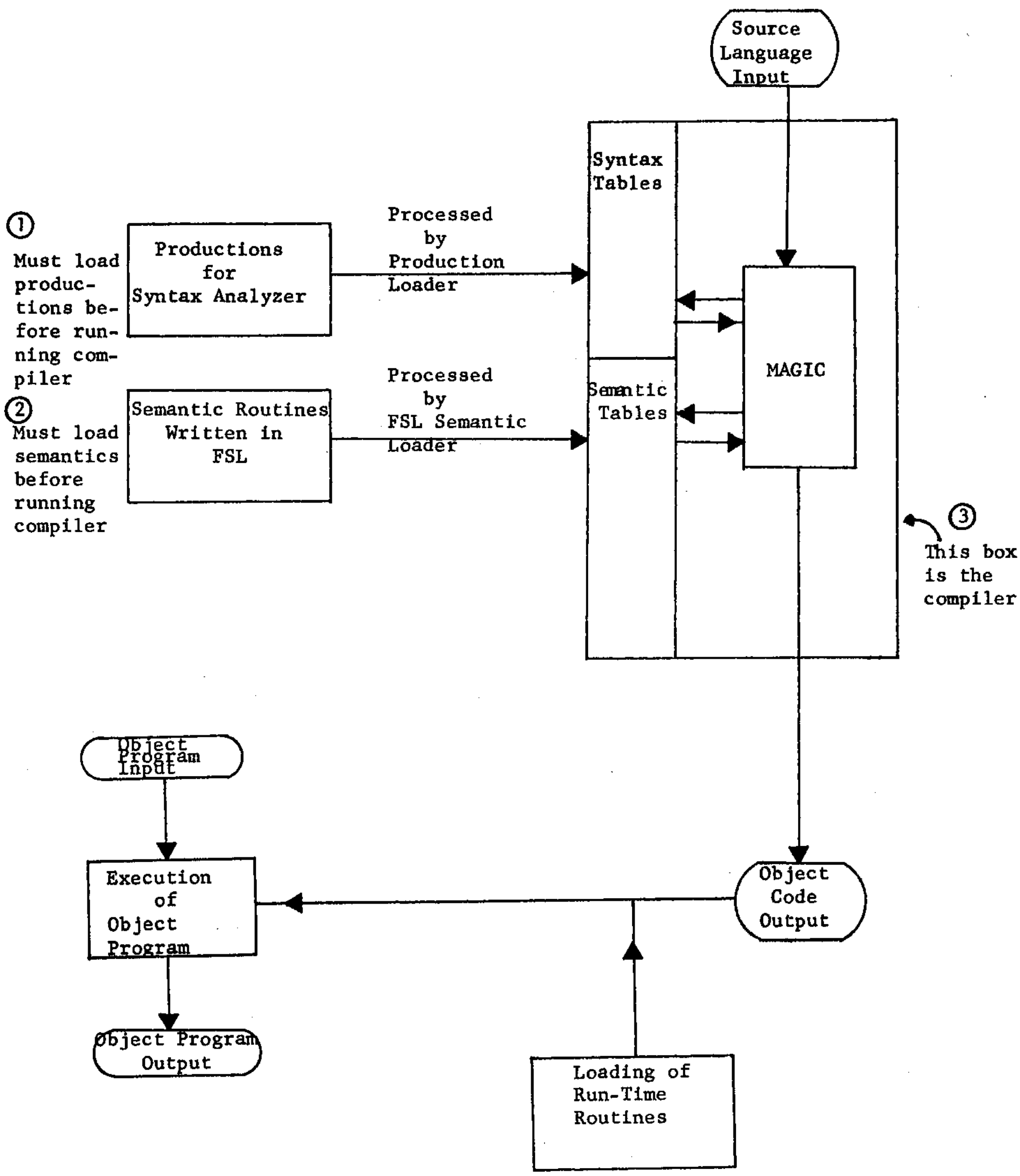

Source Language Input
Syntax Tables
① Must load productions before running compiler
Productions for Syntax Analyzer
Processed by Production Loader
② Must load semantics before running compiler
Semantic Routines Written in FSL
Processed by FSL Semantic Loader
Semantic Tables
MAGIC
③ This box is the compiler
Object Program Input
Execution of Object Program
Object Program Output
Object Code Output
Loading of Run-Time Routines

figure 1

REGULAR ALGOL

Definition: Regular Algol as discussed here constitutes all of Algol 60 excluding procedures which will be discussed separately.

DECLARATIONS

The productions are so constructed that they expect to find declarations at the beginning of blocks and in procedure headings. The first item to be processed in a declaration is the declarator. Suppose we meet REAL X, Y; in the source language. By a discrimination process which branches on the various configurations of declarators that it finds in the source language, various semantic routines are executed which set the stage for processing the variables, arrays, or switches to be declared. In the above case, REAL X, Y, the type REAL is detected, and control in the productions passes to a closed subroutine CHG with the following structure:

REAL	→	TYPE	EXEC 146	RET
INTE	→	TYPE	EXEC 147	RET
BOOL	→	TYPE+	EXEC 148	RET
LOGI	→	TYPE	EXEC 149	RET
HALF	→	TYPE	EXEC 150	RET
FORM	→	TYPE	EXEC 151	RET
SYMB	→	TYPE	EXEC 152	RET

The effect of subroutine CHG, as can be seen, is to transfer to a different EXEC routine for each of the possible types it tests against. The EXEC corresponding to a given type sets an internal variable in FSL to a value which is the FSL "title" corresponding to the syntactic "type". Thus, "types" in the syntax correspond directly with "titles" in the semantics. The type REAL in the above example would be replaced with the word TYPE in the syntax stack, and a transfer to EXEC 146 would be made causing an internal FSL variable to have its value set to the value of the title REAL.

If the declarator is a type an identifier list of variables to be declared of that type will follow. The productions are written so that all identifier lists, no matter the context in which they occur, are processed by a common subroutine of the form:

```
ID    I    | ->   |    EXEC 190     *AID
    <SG>   | ->   |    ERROR 190   AID
AID   ,    | ->   |                *ID

    <SG>   | ->   |    RETURN
```

As is seen, this production subroutine transfers control to EXEC 190 with the postfix integer corresponding to the identifier on the top of the stack. It does this for every identifier in the list. Now it so happens that identifier lists can occur in the source language in such roles as formal parameter lists in procedures, array name lists preceding bound pair lists in array declarations, and variable lists in variable declarations. In each of these different contexts it is required to process the same syntactic object, the identifier list in a different manner from the others. To accomplish this EXEC 190 is made into a variable capable of containing transfers to other EXEC's. When, in FSL, the statement XEQ 190 <-XEQ 2 is encountered, it means that the next time EXEC 190 is called, EXEC 2 will be executed. This will cause an identifier list to be processed as a variable list by the semantics. Similarly the statement XEQ 190 <-XEQ 3 will cause EXEC 190 to call EXEC 3, thus allowing an identifier list to be processed as a list of array names. By this mechanism one can treat the same syntactic construct differentially in the semantics on the basis of context.

THE SYMBOL TABLE

When variables in Regular Algol are declared they cause no code to be compiled. Rather an entry is made in a symbol table corresponding to each variable. The symbol table, declared by the FSL statement SYMB[400,4], is fixed to contain four columns which contain respectively: a postfix integer assigned by subscan to represent the identifier, an ordered pair consisting of a "type" and a "class", a machine address representing the storage location of the variable, and a context which represents the static procedure level. Each time a variable is declared a storage location pointer is incremented by one (or by two in the case of real and formula variables), and a line corresponding to that variable is entered in the symbol table. This declaration process is embedded in a block administration process which permits storage reclamation upon exit from a block by a standard push down technique (to be discussed later).

ARRAY DECLARATIONS

Array declarations are more complicated than variable declarations since not only are entries made in the symbol table, but also code is produced. During the processing of an array declaration a dimension counter is initially set to zero and is incremented each time a bound pair is encountered. The number in this counter at the termination of the count is the dimension of the array and this is known at compile time. In addition, each member of a bound pair may be an arithmetic expression so code must be produced at compile time to compute the upper and lower bounds corresponding to each bound pair. These code pieces are further embedded in code which, given a starting location, creates the head of a dope vector in the direction of descending memory addresses from that starting location. The starting location is associated with the array name by indirect addressing using the symbol table. The mechanism and form of the dope vectors is found in an article by Kirk Sattley called "Allocation of Storage for

Arrays in Algol 60" [Comm.ACM,vol.4, no.1,Jan.1961,page 60ff.]. The only departure from Sattley's mechanism is that in Formula Algol the direction of memory addresses is decreasing in the dope vectors instead of increasing. Very briefly, one saves in the head of each dope vector the dimension of the array and corresponding to each subscript a lower bound and a size [the size being the difference between the upper and lower bounds in the bound pair computed at run-time]. To access an array element $a[i_1,i_2,\ldots,i_n]$ one uses an accessing function of the form $(\ldots((i_1\text{-lowerbound}_1)\times\text{size}_1 + (i_2\text{-lowerbound}_2))\times\text{size}_2\ldots) + (i_n\text{-lowerbound}_n$. Thus, the accessing function can be computed from a knowledge of the subscripts and from the contents of the head of a dope vector. For array declarations involving lists of array names attached to the same bound pair list the mechanism of declaration is more complicated. For example, the code corresponding to the array declaration REAL ARRAY A,B,C [1:6]; would appear as follows:

```
    CLA    LOC[A]
    TRM    α
    CLA    LOC[B]
    TRM    α
    CLA    LOC[C]
    TRM    α
    TRA    θ
α:  Here we have a closed subroutine which computes the head of
    of a dope vector starting at the location given in the accumu-
    lator upon entry to the subroutine.  It looks as follows:
    ENT
    TRM    V40     [which sets switches for V41]
- - - - - - - - - - - - -
    Compute Lower Bound
    STD    T                      There are N of these code
    Compute Upper Bound           pieces, one for each of
                                  the N bound pairs.
    TRM    V41
- - - - - - - - - - - - -
    TRM    V42                [End of dope vector construction]
θ:
```

Here the transfer to V40 corresponds to meeting "[" in A,B,C[1:6], the transfer to V41 corresponds to "," and the transfer to V42 corresponds to meeting "]".

SWITCH DECLARATIONS

Upon meeting SWITCH S ← L1,L2,...,Ln in the source code the following takes place: n+1 locations are taken from array memory:

```
β   :
β+1 :
...
β+n :
```

In addition, n consecutive code pieces of the following form are produced:

```
CLA    A3
STL    β+i
TRA    A2
TRA    Li     → note:  Li is chained and therefore filled
                in prior to execution with the proper
                address.
```

Executing these n consecutive code pieces fills in the switching table. Thus, the table is filled in at the point in the program corresponding to the declaration of the switch. Later in the program, when we encounter a statement such as GO TO L[i], the following code is produced:

```
CLA    i
LXP    L,R0
TRM    V--
```

This code piece looks up the i th entry in the switching table and executes a transfer to it.

The discussion of procedure declarations, formula declarations, and symbol declarations are deferred until later.

COMPILATION OF EXPRESSIONS

Within the syntax analyzer there is a closed subroutine called the Expression Scanner whose function it is to compile code for all arithmetic and Boolean expressions in regular Algol. Later in the discussion of Formula Manipulation we will see that the expression scanner recognizes and compiles code for formula expressions,also. The expression scanner is used anytime an expression is expected in any part of the Formula Algol syntax. It is used to compile code for expressions in array subscripts, in assignment statements, in actual parameter lists, and so on.

Upon entrance to the expression scanner a discrimination is performed on the various symbols with which an expression may begin legally, and a branch is made to subsequent tests or to subroutines to compile code. For example, designational expressions must begin with IF, so if the expression scanner detects IF as the initial character of an expected expression it transfers control to a production subroutine which analyzes designational expressions. During the course of this analysis of designational expressions, arithmetic expressions or Boolean expressions may, in turn, be encountered. At the point when they are encountered control is passed back to the expression scanner. Thus, the expression scanner has been called within itself. It is important to have the expression scanner correspond to a well-defined unit of action so that it may be called by other routines any time it is necessary to recognize an expression and so that it may be called within itself. This well-defined unit of action is as follows. In the syntax stack the expression which is the input to the scanner is replaced with the single character E as the output upon return from the call. In the semantic stack corresponding to the E in the syntax stack is a description containing the type of the expression and the fact that it is to be found in the run-time accumulator. In addition, a code piece has been compiled which computes the value of the expression and which leaves the answer in the run-time accumulator.

Let us now treat some specific cases. We will examine what happens in the expression scanner when we compile code for (1) arithmetic expressions, (2) Boolean expressions, and (3) array accesses.

Single Variables

If the arithmetic or Boolean expression is a single variable this is detected immediately upon entrance to the expression scanner by a production of the form:

I | → E | *E2

The productions at E2 must now test the character following the identifier. If the following character is an arithmetic or Boolean operator, then the expression must be arithmetic or Boolean, respectively. In this case, control is transferred to a subroutine COM in the productions, which subroutine, responsible for compiling code for arithmetic and Boolean expressions. If, on the other hand, the following character is non-arithmetic or non-Boolean, then a further discrimination is required to determine what is to be done. For example, if an assignment operator "←" follows the identifier, then control passes to EXEC 9 whose responsibility it is to determine the location of the variable and to produce a semantic error if the variable was not single. If, as is also possible, the identifier is followed by the operator "[", then it is to be treated as an array identifier, and control passes to EXEC 65, which will be discussed presently. If the identifier is followed by such operators as "," ";" "THEN" "STEP" "WHILE" and others, control passes to subroutine COM in the productions. Subroutine COM, thus, lies at the heart of the compilation process for expressions. We will examine it briefly now. The routine is reproduced on pages 10a and 10b.

Subroutine COM, Arithmetic Expressions

Subroutine COM is equipped with a mechanism for sorting on the hierarchies of operators so that, for example, in the expression A + B * C, code is compiled to perform the multiplication first and the addition second, even though the order in which these operators are encountered in the syntax stack is the reverse. To accomplish this, one transfers control to subroutine COM with the syntax stack looking like E + E * |. The first production to match is production COM+5 which transfers control to H30. The productions starting at H30 will detect multiplication, division, exponentiation and unary functions SIGN,ENTIER,SQRT,EXP,LN,SIN,COS, and ABS. Thus, when * is on top of the syntax stack, the only operations that will be compiled among the elements in the second, third, and fourth positions of the stack will be those of a tighter binding power or higher hierarchy than multiplication. Note that + has a lower hierarchy than *, so nothing is compiled at this stage.

Let us now consider a complete example. Suppose we meet the statement L ←← A + B * C; in the source language. The expression scanner converts the first four characters of this statement to E ←←E + | and transfers control to subroutine COM. Here, production COM+7 matches and a transfer to H28 occurs. Nothing matches from H28 until the end, so control returns to the expression scanner which recognizes the next two characters and returns to subroutine COM with E ←←E + E * | in the syntax stack. Then production COM+5 matches the stack, control passes to production H30, nothing matches until the end of subroutine COM, control returns to the expression scanner, two more characters are recognized, and a final transfer is made back to subroutine COM. At this point the configuration of the syntax

ROUTINE FOR COMPILATION

```
COM                          L:     |                     |                        H38
 +1                          <UN>   |                     |                        H36
 +2                          ↓      |                     |                        H36
 +3                          ↑      |                     |                        H34
 +4                          NG*    |                     |                        H32
 +5                          +      |                     |                        H30
 +6                          /      |                     |                        H30
 +7                          ÷      |                     |                        H28
 +8                          -      |                     |                        H28
 +9                          <RE>   |                     |                        H26
 +10                         →      |                     |                        H24
 +11                         ^      |                     |                        H22
 +12                         v      |                     |                        H20
 +13                         CLSO   |                     |                        HA1
 +14                         <PN>   |                     |                        H19
 +15                         <OT>   |                     |                        H16
H16        E    ←+    E      <SG>   | →            <SG>   |        EXEC 112        RET
 +1        E    ←     E      <SG>   | →      E     <SG>   |        EXEC 112
                                                                   EXEC 113        COM
 +2 INSE   E    IL    E      <SG>   | →            <SG>   |        EXEC 63         RET
 +3 ALTE   E    TO    E      <SG>   | →            <SG>   |        EXEC 62         RET
 +4        |→   ↓     E      <SG>   | →      |→    <SG>   |        EXEC 197
                                                                   EXEC 207        RET
 +5             ↑     E      <SG>   | →            <SG>   |        EXEC 198
                                                                   EXEC 207        RET
 +6 E      IS   NOT   E      <SG>   | →            <SG>   |        EXEC 108        RET
 +7 E      IS   ALSO  E      <SG>   | →            <SG>   |        EXEC 109        RET
 +8        E    IS    E      <SG>   | →            <SG>   |        EXEC 176        RET
H19        E    INST  E      <SG>   | →      E     <SG>   |        EXEC 85         COM
HA1        E    CLSO  E      <SG>   | →      E     <SG>   |        EXEC 77         COM
 +1             CLSO  E      <SG>   | →      E     <SG>   |        EXEC 80         COM
H20        E    v     E      <SG>   | →      E     <SG>   |        EXEC 105
                                                                   EXEC 114        COM
H22        E    ^     E      <SG>   | →      E     <SG>   |        EXEC 105
                                                                   EXEC 115        COM
H24             →     E      <SG>   | →      E     <SG>   |        EXEC 116        COM
H26        E    <     E      <SG>   | →      E     <SG>   |        EXEC 100
                                                                   EXEC 117        COM
 +1        E    >     E      <SG>   | →      E     <SG>   |        EXEC 100
                                                                   EXEC 118        COM
 +2        E    NL    E      <SG>   | →      E     <SG>   |        EXEC 100
                                                                   EXEC 119        COM
 +3        E    NG    E      <SG>   | →      E     <SG>   |        EXEC 100
                                                                   EXEC 120        COM
 +4        E    ≠     E      <SG>   | →      E     <SG>   |        EXEC 100
                                                                   EXEC 121        COM
 +5        E    =     E      <SG>   | →      E     <SG>   |        EXEC 187
                                                                   EXEC 122        COM
H23        E    +     E      <SG>   | →      E     <SG>   |        EXEC 100
                                                                   EXEC 123        COM
 +1        E    -     E      <SG>   | →      E     <SG>   |        EXEC 100
```

```
H30    E   »     E   <SQ>  1  •«••    E-   <SG>  |
• 1    E   /     E   <SG>  1          E •  <SG>  |
H32        NG*   •E  <SQ>  1          E    <SG>' |
H34    E-        E   <SQ>  1 •«•-     E    <SG>  |
H36        SIGN  •g. <SG>  1          E'   <SG>  |
 + 1       ENTI  E   <SQ>  1          E    <SG>  |
 + 2       ARCT  •E  <SG>  I          E    <SG>  |
 + 3       SORT  E   <SQ>  1          E.'  <SG>  |
 + 4       EXP   •E  <SQ>  1  «•      E    <SG>  |
 +5        UN    E   <SQ>  1  •»      E    <SG>  |
 + 6       COS   E   <SQ>  1  *       E    <SG>  |
V + 7      SIN   E   <SG>  1  •«••    Ei   <SG>  |
• +8       ABS   E   <SQ>  1          E<   <SG>  |
: +9       *     E   <SS>  I  *       E    <SG>  |
H38    E   L:    E   <SG>  f-«»-      E;   <SG>  |
                   -<SG>"
```

```
EVEC 124    COM
EXEC 100
EXEC 125    COM
EXEC 100
EXEC 126    COM
EXEC 107
EXEC 127    COM
EXEC 100
EXEC 128 -  COM
EXEC 107
EXEC 129-   COM
EXEC 107
EXEC 130    COM"
EXEC 107
EXEC 131"   COM
EXEC 107
EXEC 132-   COM-
EXEC 107
EXEC 133    COM
EXEC 107
EXEC 134    COM
EXEC 107
EXEC 135 -  COM
EXEC 107
EXEC 136 •  COM
EXEC 107
EXEC 137-   COM
EXEC 107
EXEC 138'   COM
EXEC 87     COM
RETURN  "
```

stack is

$$E \leftarrow \leftarrow E + E * E \; ; \; | \; .$$

Here the metacharacter <OT> matches the semi-colon on top of the stack at production COM+15, and control passes to production H16. The first production to match the stack is production H30. This is the first instance of any compilation in the processing of the statement. All previous actions up until this point have consisted of postponements. The compilation is accomplished by transfers to EXEC 100 and to EXEC 125, which compile code to multiply B and C. In the case of arithmetic operands

```
CLA B
MPY C
```

is constructed. In the case of formula operands, code to construct the formula tree $\overset{*}{B \; / \; \backslash \; C}$. The semantic routines used to accomplish this, test the types of the operands and compile the appropriate code. At the completion of this compilation the syntax stack is altered to look like $E \leftarrow E + E;$ because the terminal E * E has been replaced by a single E, as is seen in production H30. The semantic routines also set the description of the topmost E to contain the type of the expression and the fact that it is in the run-time accumulator. Control now passes back to the beginning of subroutine COM for another iteration of the process. Subroutine COM will be seen to reenter itself iteratively until the entire expression is consumed, until code for it has been compiled, and until its external representation in the syntax stack has been replaced by E in the case of pure expressions and nothing in the case of statements, some of which are handled by subroutine COM.

We are now at the point where the syntax stack looks like $E \leftarrow \leftarrow E + E; \; |$ and where we have reentered COM. On this pass production COM+15 matches and passes control to H16 where successive productions fail to match the syntax stack until production H28, at which point E + E is compiled by EXEC 100 and EXEC 123. The routines in MAGIC at compile time inspect the descriptions of the operands and are smart enough in this case to compile

```
CLA  B
MPY  C
ADD  A
```

in the case of arithmetic expressions since the description of the second operand in LEFT2 contains the information that the result of the current compilation is in the run-time accumulator. Again the semantic routines analyze the types of LEFT2 and LEFT4 to determine whether code should be compiled to add numerical expressions or to add formula expressions. After compilation the stack configuration is changed to E $\leftarrow \leftarrow$ E; and control passes back to the beginning of subroutine COM. On this final trip through subroutine COM production H16 constructs code to perform the assignment of LEFT2 to LEFT4 and subroutine COM is exited with only the semi-colon remaining in the syntax stack, the statement having been consumed entirely. In the case of expressions, rather than statements, an E is left upon exit in the RIGHT2 with its semantic description set to contain its type and the fact that it resides in the run-time accumulator.

The Administration of Temps

During the compilation of arithmetic expressions and Boolean expressions it is occasionally necessary to use temporary storage to save the partial result of a computation while another partial result is being prepared in the accumulator. In Formula Algol temps come from normal storage where they may participate automatically in the mechanisms of recursion. Temps are reclaimed when a block is exited just as is normal storage private to the same block. All temps are used only once per block and then thrown away. This is a trade off of a small amount of space for a large amount of compile time efficiency since no stacking and no memory system need be used to administer which temps are assigned and which are free.

Boolean Expressions

Boolean expressions are compiled in exactly the same manner as arithmetic expressions by subroutine COM. The only difference is that different binary and unary operators are involved and that the types of the operands are different. The semantic routines perform tests to ascertain that the types of operands involved in Boolean expressions are Boolean and not arithmetic. Likewise, type checking ascertains that operands in arithmetic expressions are not Boolean, and that operands on the right and left sides of assignment arrows are legal. If illegal combinations are detected, semantic errors or "Faults" are printed out at compile time.

Array Accesses

Suppose we are asked to compile the following statement:

$$B[I] \leftarrow A[I+1,J+K,I] + 3;$$

We immediately see that there are two cases to consider. The array element on the left hand side of the assignment statement is to be stored into whereas the array element on the right is to have its value accessed. In the first case we need code to produce an address. In the second case we need code to produce a value. To discriminate between the two cases we use the fact that the array element on the left hand side can be detected upon entrance to the Statement Scanner [to which control is transferred in the syntax analyzer at the beginning of the analysis of every statement] whereas the second array element on the right hand side will be processed by the expression scanner. Thus, embedded in the statement scanner at the very beginning is the following structure:

```
S1    |- I |→      E  |     *S2
S2    E  [ |             Call to an EXEC to produce
                               LXP  0  0,R0
```

In the other case in the expression scanner we have

```
E2    E [ |                          Call to an EXEC to produce
                                              LXP  0  1,R0
```

Then both cases converge by producing a transfer to a subroutine in the syntax analyzer to process expression lists [which are subscript lists for the array elements]. At the time of this convergence another instruction is inserted in the code compiled:

```
LXP  0  k,R0                         where k = 0,1 for the
                                     left and right sides
TRM     V44                          respectively.
```

The productions that process the subscripts compile the following code:

```
LXP  0  k,R0
TRM     V44

[code piece to compute first subscript and to
 leave result in run-time accumulator]

TRM     V45

[code piece to compute second subscript and to
 leave result in run-time accumulator]

TRM     V45

..........

[code piece to compute last subscript and to
 leave result in run-time accumulator]

TRM     V46
```

Here

V44 Saves the contents of R0 in a switch available for later use by V46 which will need to know whether an address or a value is needed, and administrates a push down stack for array subscripts for array calls within array calls.

V45 Constructs code for partial accessing of an array element using the information in the head of the dope vector according to the formula (subscript - lower bound)* size.

V46 Looks at the switch set by V44 and knows whether to produce code accessing the address or the value of the array element.

Hence, the code compiled for the statement

B[I] ← A[I+1, J*K,I] + 3;   is as follows:

```
LXP  0  0,R0
TRM     V44
CLA     I
TRM     V45
TRM     V46
STD     T?
LXP  0  1,R0
TRM     V44
CLA     I
ADD  0  I
TRM     V45
CLA     J
MPY     K
TRM     V45
CLA     I
TRM     V45
TRM     V46
STD  1  T1
```

Push Down Mechanism in Formula Algol

The following mechanism for pushing down, saving, and restoring variables is used throughout Formula Algol at run-time. For example, it is used in the print routine, in the evaluation routine, and in all routines that call themselves or each other recursively. It is, therefore, important to know about it and it is introduced here for that reason.

There is a region of safe cells S0,S1,..., S100, and, in addition, a long push down stack, the top of which is saved as an address in index register R6. There are also two routines, V25 and V26, which push and pop this stack, respectively. Suppose the first N cells in the S region contain information which is to be saved. The number N and a return address to be transferred to upon pop up are communicated as input parameters to V25.

The number of locations of the S region to be saved is inserted in the index register R1, and the location to return to is inserted in index register R0. Then V25 is called. This transforms the push down stack by appending the contents of the first N cell in the S region to the stack, and by adding a word pair containing the following three items: a chaining address for use in popping up the stack, the return address, and N. The following figure depicts this transformation.

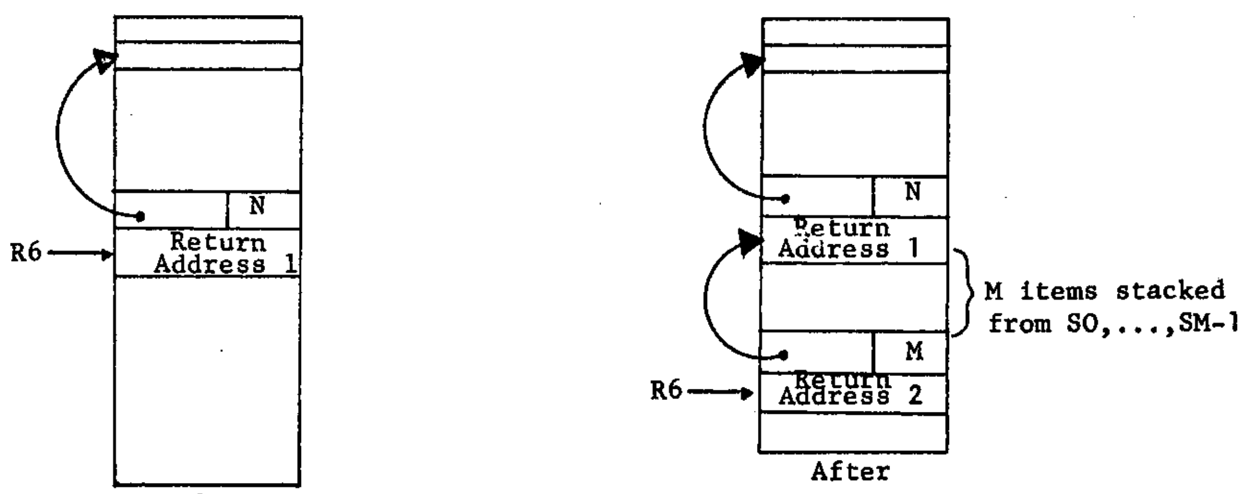


figure 2

Executing V26 restores the top N variables on the push down stack to the first N cells in the S region, pops up the stack by changing the contents of R6, and executes a transfer to the return address saved on the stack. Thus, recursive exits = TRA V26.

Conditional Expressions

Suppose we wish to compile conditional expressions of the form:

IF B THEN E1 ELSE E2 ;

This is accomplished by a subsystem of the productions which has the following structure:

```
IF     |                     |        *E1 (in statement scanner)
IF   E   THEN | →    THEN    |  EXEC 30   *E1
THEN E   ELSE |      ELSE    |  EXEC 31   *E1
ELSE E   ;    |      ;       |  EXEC 32
ELSE E   END  |      END     |  EXEC 32
```

Here EXEC 30 produces code to push a flad.

    PUSH [FLAD1,0]; CODE(¬LEFT2 → JUMP[FLAD1]);

This creates code to transfer to an as yet undefined address if the Boolean expression of LEFT2 is false. In EXEC 31 we have to create code to correspond to case when the first expression has been computed and when we want to jump around the code to compute the second expression. To do this we need a second flad. The code for EXEC 31 looks as follows:

    PUSH[FLAD2,0]; CODE(JUMP[FLAD2]); ASSIGN[FLAD1];

The last statement assigns the current codeloc to be the address to which the transfer is made in the event that the Boolean condition is false. Finally, at EXEC 32 all that remains to be done is to assign flad2, which will be the address to which the transfer is made after computing the first expression in the conditional. EXEC 32 looks like:

    ASSIGN[FLAD2];

The code produced from this process corresponding to the entire conditional statement then looks as follows:

```
    FUO   TRUE
    TRA   α
    [ codepiece to compute E1 ]
    TRA   β
  α [ codepiece to compute E2 ]
  β whatever else is compiled next in the program
```

The situation for conditional expressions not involving ELSE is much simpler.

We just have a production which looks like

THEN E ; | → ; | EXEC 33

where in EXEC 33 we do

ASSIGN [FLAD1]

to create a jump around the code which computes the value of the expression or which executes a statement. Because conditionals may be nested it is important to have flads which are push down stacks. Actually EXEC 30 is a bit more complicated than indicated here because of the necessity of merging with Formula manipulation. The Boolean expression in LEFT2 could possibly be an EVAL expression which upon execution at run-time could either collapse to a Boolean value or could fail to collapse to such. To handle this situation at compile-time one sets the type of an EVAL expression to "TRUMP" and EXEC 30 tests for type TRUMP. Upon finding type TRUMP code is produced to transfer to a run-time routine to check the type of the result left by the EVAL expression. If the type is Boolean, then the situation is the same as that explained above. If the type is not Boolean, then a run-time error is printed.

<u>Designational Expressions</u>

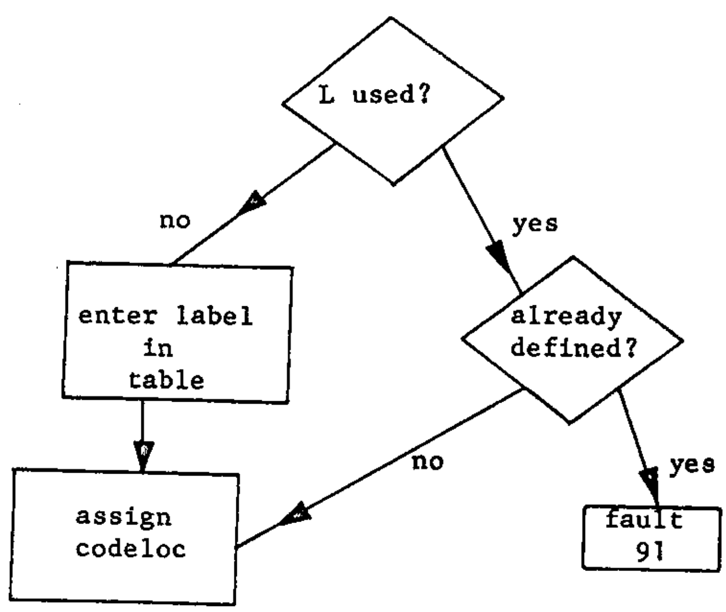
Statements may, of course, be labelled, and, therefore, upon entrance to the statement scanner, whose job it is to analyze all possible ways in which a statement may begin legally, the presence of L : is detected by a production of the form

E : | → | EXEC 91 *S1 .

As is seen the E : is eliminated from the syntax stack and the statement scanner is reentered. EXEC 91 is, therefore, totally responsible for processing the labels that occur attached to statements. References in designational expressions may be of two types: (1) those which transfer to an undefined label which has not yet occurred in the source program, and (2) those which transfer to a label already defined which has occurred previously. The compiler must discriminate between these two cases. The first requires that all references to the undefined label be chained. The second merely requires compilation of a transfer from information given in a label table, the stratagem being to store in the label table the address of code location corresponding to the beginning of the labelled statement once such information becomes available during the compilation. In Formula Algol the label table has five pieces of information in it (in contrast to the symbol table, which has four). The name of the label table is LAB, and we might picture its structure as follows:

```
LAB [     postfix integer for the label,or switch,
          title which is either LABEL or SWITCH, loca-
          tion in code corresponding to label, level,
          tag = 0 for defined and 1 for undefined  ]
```

We now turn our attention to EXEC 91. A flow chart for it is as follows:



The FSL translation of this flowchart is:

$T \leftarrow$ LAB[LEFT2....\$];

SIGNAL $\rightarrow T = 0 \rightarrow$ FAULT 91:

```
LOC[ LAB [ 0,,,, $ ] ← 0 ;

ASSIGN [ LOC [ LAB [ 0 ,, $,, ]]] $  :

T ← CODELOC ;  ENTER[LAB ; LEFT2,LABEL,T,LEV,0]$
```

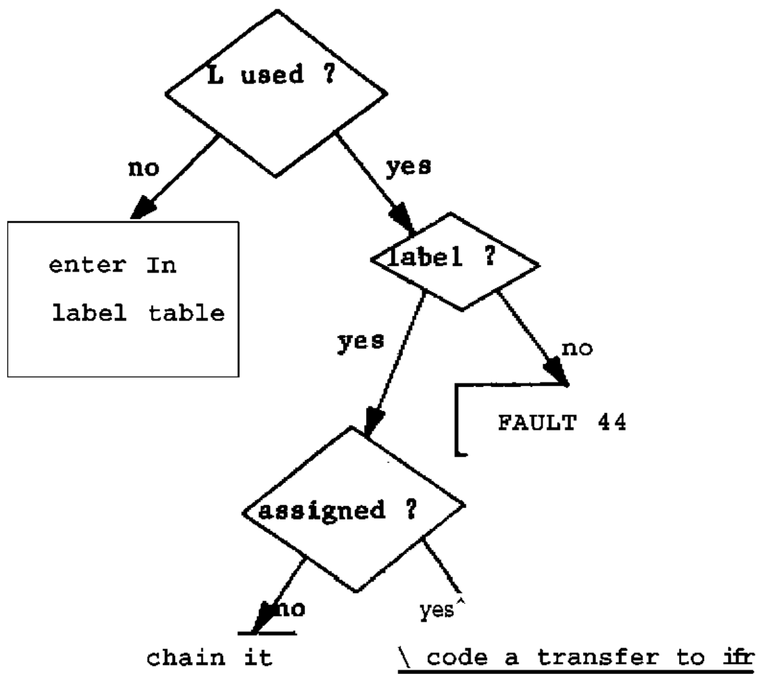
The main idea of the FSL code is this. T is a temporary into which the extracted tag is placed. During the extraction if the postfix identifier LEFT2 can't be found in the table LAB, the SIGNAL is set false; otherwise it is set true. A test is next made on SIGNAL, and if it is true, then the postfix integer LEFT2 was already in the table. It must, therefore, have been either used or defined. If it was defined, i.e. if T = 0, then this is the second time the label is being defined, so we print FAULT 91; otherwise we set the tag in the line where it was registered undefined to 0 to denote that it has just become defined. We further place the current code location in the third column of the table. In the event that the label was not in the table, then we enter the postfix integer, the current code loc, a title LABEL, a tag of 0, and the current level into the label table.

Now suppose we have the statement GO TO L where L is a label rather than a switch. In the productions we will find the following subsystem:

```
(for switches)   GO  TO L   [    |          | * E1

(for labels  )   GO  TO L <SG>   | → <SG>   | EXEC 44 *S1
```

The second of these productions completely eliminates the GO TO L statement from the stack and transfers to EXEC 44. A flow chart of EXEC 44 is as follows:



The FSL code for this is:

```
'ALPHA'
   T «-LAB [ LEFT2, , , , $ ] ;
   SIGNAL -»
      LAB [ 0,$, , , ] = LABEL -»
   COMT2 +-LOC [ LAB [0, , $ , , ] ;
   T = 0 ->
      COMT3 <-<COMT2>;CODE (JUMP[COMT3 ] ):
   CODE ( JUMP [ CHAIN [COMT2] ] ) $ :
          FAULT 44 $ :
   ENTER [ LAB; LEFT2, LABEL,0,LEV, 1];
   JUMP [ ALPHA ] $
```

A verbal analysis of this FSL code is as follows. First one looks up the label LEFT2 in the label table and extracts the tag if it is there. If the label is there, SIGNAL is set true and the tag extracted is placed in T. Otherwise SIGNAL is set false. Suppose the label was in the table and that the tag has been placed in T. This means the label was used, and the tag will tell whether the label is defined or undefined. We first check to see if the title of the postfix integer found was LABEL. If it wasn't we print FAULT 44. If it was we extract the location in the table of the place where the code location is to be stored and store this table location in COMT2. Then we test the tag to see if the label was previously defined. If it was, we extract the code location from the table (which was entered when the label became defined) and place this in COMT3. Then we code a transfer to COMT3. If, on the other hand, the label was undefined,we must chain an undefined reference to the position in the table where the location will later be entered. In the event that SIGNAL was set false, the label wasn't in the table, so the last lines of the FSL code enter the label in the table and reenter the routine to process the label in the same fashion as defined labels. One should notice at this point that the ASSIGN statement on the top of page 20 assigns all undefined forward references to the label,if any,by means of the chain set up in EXEC 44.

A final topic in the discussion of designational expressions is the processing of statements involving transfers to switches. E.G. GO TO SW [K + 4]; A production of the form

```
GOTO  E  [  E  ]  | →      | EXEC 45   *S1
```

handles all such designational expressions. Since switches must be declared, they are always in the label table, otherwise it is a semantic error. We have already treated the declaration of switches in the discussion of declarations, and we saw there that switch declarations cause code to be compiled which, when executed, builds up a switching table in the space used for dynamic array storage. This switching table is of the form:

```
T :        n
      TRA  L1
      TRA  L2
      . . . .
      TRA  Ln
```

Thus, EXEC 35 has the following structure:

[some tests to see that things are declared, etc.] →

T ← LAB [LEFT4 , , $, ,] ;

CODE(Y1 ← LEFT2 ; ACC ← LEFT4 ; JUMP[<X35>])

This produces code to place the value of the subscript expression in the run-time cell Y1, to place the location of the switching table in the accumulator, and to mark transfer to a routine X35. This routine is executed at run-time and compares the value of the subscript expression with the number n stored in the head of the switching table to see if the subscript has exceeded the switching table dimension, and if it hasn't, executes the appropriate transfer. If it has, it prints a run-time error.

This completes the discussion of designational expressions.

FOR STATEMENTS

In the processing of for statements the crucial mechanism concerns the compilation of code to correspond to each of the several possible for list elements. This is done by a case analysis. The cases are:

A. E1,

B. E2 WHILE E3

C. E4 STEP E5 UNTIL E6

D. E7 STEP E8 WHILE E9

For these cases, code is produced as follows:

CASE A

```
I ←E1      (I is the control variable in these examples)

TRM  S     (here S is a closed subroutine corresponding
            to the body of the for statement)
```

CASE B

```
α I ←E2                         (We are using a mixture of Algol

  IF ¬ E3 THEN GO TO β           and machine language to describe

  TRM S                          the code.  Substitute code for

  TRA α                          the Algol if you want to be pure.)

β . . .
```

CASE C

```
   I ←E4

   TRM β1

   TRA β2

β1 ENT               (compute step)

   T ←E5

   TRA  I  β1

β2 IF (I-E6)*SGN(T) > 0 THEN GO TO  β3    (exit condition)

   TRM S

   TRM β1

   I ←I + T

   TRA β2

β3 ...
```

CASE D

```
   I ←E7

   TRM  β1

   TRA  β2
```

```
β1 ENT
   T ←E8
   TRA 1 β1
β3 TRM β1
   I ←I + T
β2 IF ¬E9 THEN GO TO δ
   TRM S
   TRA β3
δ  . . .
```

Here we will discuss the case where we produce code for the STEP UNTIL case (case C). The others will not be discussed as the reader versed in FSL will be easily able to generalize the process for himself.

Let's take a specific example:

FOR I ←3 STEP 4 UNTIL 19 DO PRINT(I) ;

Upon seeing FOR as the initial character of a statement, the statement scanner transfers control to the expression scanner to recognize and to process the control variable. The expression scanner reduces the control variable to E and scans the assignment arrow ←. Control is then transferred to a utility routine of the expression scanner, routine E5, whose second production is

FOR E ← |→ FOR E ↞ | EXEC 211 *E1

This production converts the single assignment arrow ← to a double assignment arrow ↞ representing a destructive store. EXEC 211 finds the location of E and saves it for later use in the processing of each for list element. Control then returns to the expression scanner. The expression scanner picks up the lower bound for the for variable, compiles, by means of subroutine

COM the assignment E $\leftarrow\!\!\leftarrow$ E, producing the code

I $\leftarrow$ 3

then following this a STEP is picked up upon return from COM and control is transferred to utility routine F10 where the production

FOR STEP | $\rightarrow$ STEP FOR | EXEC 40 F10A

matches. EXEC 40 is as follows:

```
PUSH[FLAD1,0]; PUSH[FLAD2,0]; CODE(MARKJUMP[FLAD1];
JUMP[FLAD2]); ALFA ← CODELOC; ASSIGN[FLAD1]; TALLY[CODELOC];
```

This produces the following code:

```
      TRM  β1
      TRA  β2
   β1 ENT
   →
```

The production at F10A inserts E $\leftarrow\!\!\leftarrow$ into the stack.

F10A <SG> |$\rightarrow$ <SG> e $\leftarrow\!\!\leftarrow$ | EXEC 60 *E1

EXEC 60 assigns RIGHT2 the semantics of a temporary and stores its location and description in the semantic stack. Control then returns to the expression scanner which scans the step function and compiles an assignment into the temp inserted into the stack by the production F10A. Next the UNTIL is detected, and control transfers to F15, where the following production matches:

STEP FOR UNTIL | $\rightarrow$ UNTIL FOR | EXEC 41 F15A

EXEC 41 is as follows:

```
CODE(JUMP[<ALFA>]); ASSIGN[FLAD2] ;
```

The following code is thus added to the codestack:

```
      T ← 4
      TRA 1 β1
```

[The reader should refer to the example of code on page 24 for Case C to see how this code fits in with the previous code]. At F15A a - E is inserted into the stack by the following production:

F15A <SG> | → <SG> E - | EXEC 61 *E1

Here EXEC 61 assigns the semantics of the control variable to E and puts its location in the semantic stack. This allows the expression scanner to compile (I-19) for use in determining the termination conditions for the for statement. This allows the code for IF (I-19)*SGN(T) to be produced automatically using the mechanisms of subroutine COM. Finally, when control is transferred from subroutine COM back to the expression scanner, and when the expression scanner picks up DO on top of the stack, control is passed to production subroutine F31, where the following production matches the stack:

F31 UNTIL FOR E DO | → DO ⊢ | EXEC 26

EXEC 26 is the final EXEC in the processing of the for statement (except, of course, for those responsible for making the body of the for statement a closed subroutine). EXEC 26 looks like this:

PUSH [FLAD1,0]; CODE(T*LEFT2 > 0 ; JUMP[FLAD1]; MARKJUMP[FLAD2];
MARKJUMP [ALFA] ; CODE(TT ← TT + T) ; CODE (JUMP[BETA]) ;

Here MARKJUMP[FLAD2] produces TRM S , MARKJUMP[ALFA] produces TRM β1 and CODE (TT←TT+T) produces I ← I + T where TT has been assigned the semantics of I, the control variable, and where T has been assigned the semantics of the step expression. Finally, CODE (JUMP[BETA]) produces a transfer TRA β2. Here β2 was assigned in EXEC 61.

This completes the discussion of for statements.

PROCEDURES IN FORMULA ALGOL

We will first discuss procedure calls. Suppose we meet the procedure statement:

P (A , B + 1 , C * D) ;

in the source language text. The statement scanner picks up the procedure identifier with a production of the form

```
51    I | ->       E  |          *S2
52  E ( |             | SUBR COL S2A
```

Thus, control is transferred to production subroutine COL, where the list of actual parameters is processed. The expression scanner contains a nearly identical subsystem of productions of the form

```
E1    I | ->       E  |          *E2
E2  E ( |             | SUBR CAL E2A.
```

This subsystem transfers control to production subroutine CAL. The difference between subroutine CAL and subroutine COL is that CAL corresponds to the use of a procedure as an operand in an expression, whereas COL corresponds to the use of a procedure as a statement. These two routines allow control to be returned to the expressions scanner from CAL and to the statement scanner from COL after the list of actual parameters has been processed in each case. Upon entrance to both CAL and COL a transfer is made to EXEC 11, which compiles a transfer around the thunks which will be inserted in the code corresponding to the actual parameters, and which marks the thunk stack ACT with a special marker to delimit the thunks corresponding to the current actual parameter list being processed. The code corresponding to the procedure call P (A,B+1,C*D) will look as follows:

```
      TRA   α      ←(note: no code is produced for A since it is
    β CLA   β        a single identifier whose location can be used)
      ADD   1
      STI   VCP    (VCP is a special location known to the run-time
      LXP   VCP,R0  routines that process procedure calls)
      TRA   V204
    γ CLA   C
      MPY   D
      STD   VCP
      LXP 0 VCP,R0
      TRA   V204
      003   γ       (These three quantities are the three thunks
      003   β        corresponding to the three actual parameters
                     in the procedure calls. The numerical codes
      001   LOC[A]   001 and 003 tell what type of thunk is involved.)
    α TRM   V201    (Run-time routine V201 handles procedure calls.
      000   P        From the mark of the call one knows where to
                     find the thunks by subtraction.)
      CLA   1,R-1   (R-1 is a fixed index register which contains
                     values from V201. This command is compiled if
                     the value of the procedure is desired.)
```

THUNKS

During the actual parameter scan transfers are made to EXEC 11 by productions of the following form:

```
E  ,  | →          | EXEC  12   *E1
E  )  | →          | EXEC  12
```

Here EXEC 12 creates a thunk corresponding to each actual parameter and stacks it in a compile time stack called ACT. When all of the actual parameters have been scanned, i.e. when) is hit in the syntax stack, all thunks are unloaded into the code and a return is made via CAL or COL to E2A in the expression scanner or to S2A in the statement scanner to compile a call to the procedure. Of course there can be arbitrary nesting of calls in the actual parameter list, and so the stack ACT has to be set up to handle this possibility. Stack markers are used for this purpose. A marker is pushed onto the stack when a new list of actual parameters is encountered, and when dumping the thunks into code one pops back to the previous marker. The table for the various types of thunks is as follows:

1	m m m	m m	n n n n n	dynamic variable	loc = M+<N>
0	$\frac{1}{2}$ 0 0	n n	n n n n n	signed integer	val = ± N
0	0 0 1	0 b	n n n n n	variable or abcon	loc = bN
0	0 0 2	0 b	n n n n n	array	head = bN
0	0 0 3	0 0	n n n n n	code piece	start = N
0	0 0 4	λ λ	n n n n n	label	{dest = N, target level = Λ}
0	1 0 5	m m	n n n n n	formal parameter	{position = M, procedure = N}
0	0 0 6	0 0	n n n n n	procedure	name = N
0	0 0 7	0 0	n n n n n	switch	name = N

Having compiled the thunks and having inserted them in code corresponding to the actual parameter list one is now in a position to compile the procedure calls. This must be accomplished by a chaining algorithm which is sensitive to static block levels. When the calls are encountered we chain them through the code and upon exiting a block we assign all calls within that block that are still in the chain. For example: Given a piece of source language text with the structure

```
BEGIN
   PROCEDURE P ...
       BEGIN
         Q( ) ;  F(Q) ;
       END
   PROCEDURE Q
       BEGIN
       END
END
```

Here the call of Q comes before the declaration of Q, so forward chaining is needed. The first call Q( ) causes two words to be inserted in code (at position α in the code sample on page 29) which two words have the following structure:

TRM ERROR

[static block level] [chain address or chain end]

Similarly, the use of Q as an actual parameter in F(Q) causes a one word codepiece of the form [bit to distinguish one word from two word case] [static block level] [chaining address or chain end]. After the chained calls become assigned by means of an assignment algorithm we shall discuss presently, the word pair case looks like this:

TRM V201

000 address of Q (i.e. address of first word of code corresponding to Q.)

And the single word case is a thunk which looks like this:

006 address of Q

Some tables and stacks are used to provide an association function between procedure names and the chains of their calls. The table is called CRADLE and has procedure identifier (or their post fix integers) in the first column and has chain heads in the second column.

CRADLE

P	o
Q	
F	

chain through code of all calls of P.

The stack called LADLE stacks all calls on procedures which occur in a block. Upon entrance to the block a zero is stacked in LADLE, and each call is stacked as part of a word pair in this stack. At the end of a block the assignment algorithm assigns all calls corresponding to the procedures in the stack and terminates upon reaching a zero. The assignment algorithm extracts the chains from the table CRADLE and by arithmetic comparison on the block level information contained in each call in the chain can determine whether a call should be assigned at that block level or not. All assigned calls are removed from the chain and those which cannot be assigned are left in the chain. These remaining calls in the chain may then be assigned at higher block levels.

To enter things in the chain corresponding to a given procedure there is a routine called HEAD (I). HEAD finds or creates an entry in CRADLE. If the identifier is found in the first column it gives the location of the head of the chain found. If the identifier is not found it puts it there and gives the location of the head of a chain which it creates. The following FSL code does this:

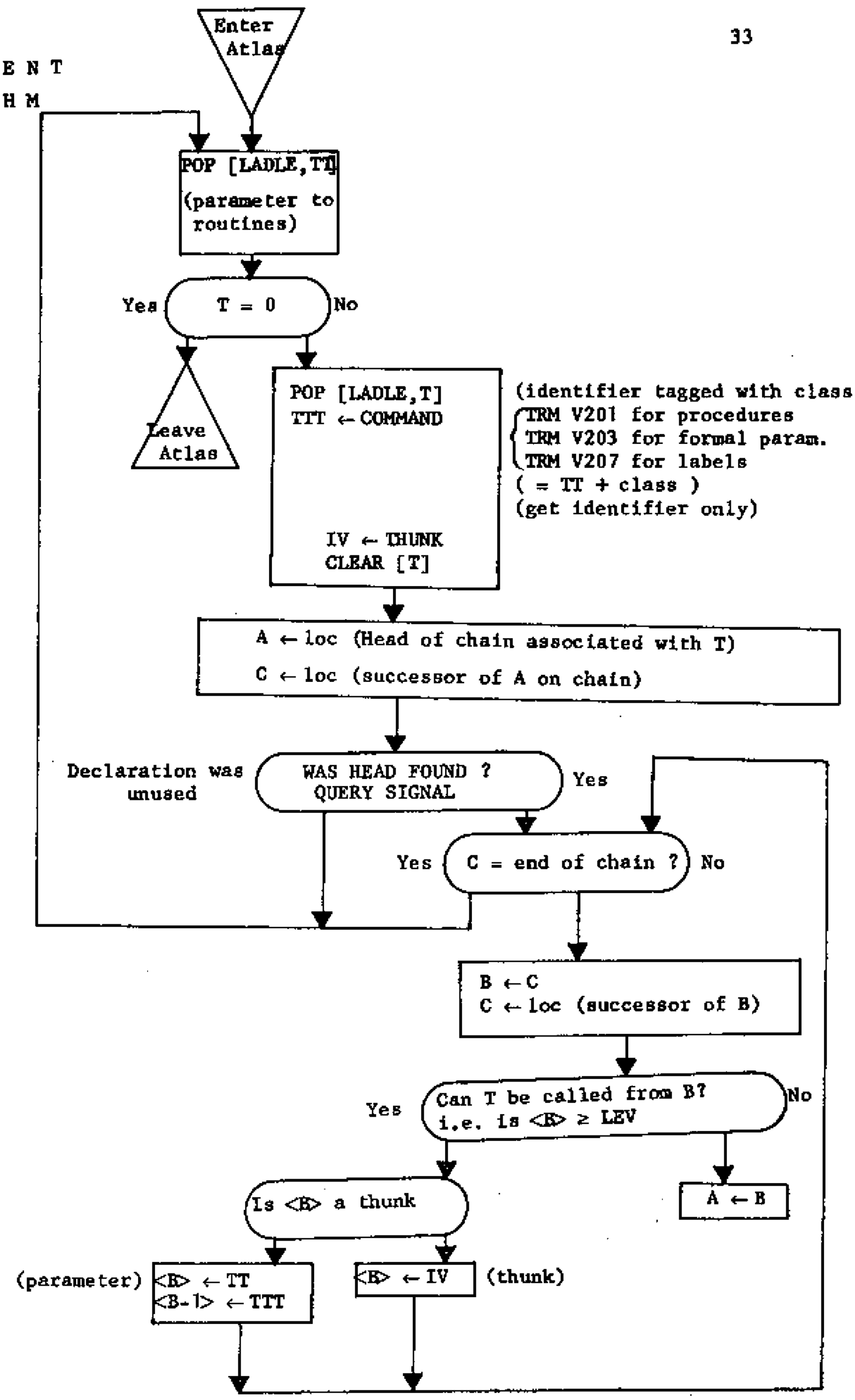
```
  T ← LOC [CRADLE [LEFT2, $ ] ] ;
¬ SIGNAL → ENTER [CRADLE;LEFT2, CHAINEND ];
  T ← LOC [CRADLE] - LENGTHOF(-CRADLE) ;  (this puts the location
                                           of the head of the chain
                                           in T)
```

We can now use this routine to create the chain corresponding to a call. This is done by a routine CALL(I) which looks as follows:

```
MARKJUMP[CALL(I)]; <CODELOC> ← CHAIN(<T>) + LEVEL;
TALLY[CODELOC]; TT ← <T> ; <T> ← CODELOC;
<CODELOC> ← T T  +  LEVEL ; TALLY[CODELOC] ;
```

This routine CALL(I) is executed for procedure calls both as expressions and as statements and for procedure identifiers occurring as actual parameters. It remains to discuss the assignment algorithm executed upon block exit. A flow chart for this appears on the next page.

A S S I G N M E N T
A L G O R I T H M



ASSIGNMENT ALGORITHM

This assignment algorithm is realized by a routine called ATLAS, and its broad strategy is this: ATLAS pops the successive procedure names from the stack LADLE and processes these one by one. When it comes to a zero in LADLE the processing is finished. For each procedure name in LADLE it looks this procedure name up in the association table CRADLE and finds the chain of calls on that procedure. It then steps down the chain making arithmetic comparisons on each item in the chain to determine if a call on that particular procedure. It then steps down the chain making arithmetic comparisons on each item in the chain to determine if a call on that particular procedure is legal at the current block level. It then assigns those which are legal by substituting in the code pair TRM V201 followed by the procedure address [or in the case of thunks a procedure address with the appropriate thunk code]. Those calls that get assigned are deleted from the chain. Those that are not assigned remain in the chain to be assigned at higher block levels with some possibly different meaning.

In a similar fashion ATLAS assigns labels and formal parameters. These items are also stacked in LADLE and the same chaining algorithm with minor variations is used on them. Likewise, with minor variations from the case discussed above, they are assigned by ATLAS.

Having discussed procedure calls we now turn to procedure declarations.

The code corresponding to a series of procedure declarations looks as follows:

```
        TRA  θ
     α  Context of Procedure
        Block Level, Amount of Storage Required }- ->HEAD

        [          ]  - - - - -> Code for Procedure Body

        TRA  V202        (<-resets storage, finds last context
                            and returns to where came from)
        Other Procedure Declarations of same form as above.
     θ  ....
```

As is seen, the TRA θ constitutes a single jump around a series of procedure declarations. Suppose we want to compile code for a procedure declaration that starts REAL PROCEDURE P(A,B);. In the productions the type REAL will be picked up by a production of the form

<TYPE> | | SUBR CHG   * SEC.

Subroutine CHG, which was discussed on page3,sets an FSL variable with a "title" corresponding to REAL, and it substitutes the word TYPE for the word REAL in the syntax stack. Thus, control passes to a production SEC with the syntax stack looking like TYPE PROCEDURE x |. At SEC the following production matches:

SECX   PROC | | EXEC 159   *PRI

In EXEC 159 we save the current contents of STORLOC by pushing it onto a stack, and we set up relative addresses in STORLOC by initializing it to 1. Thus, we write, in FSL,

PUSH[STAB, STORLOC]; STORLOC ← 1 ;

Also, in EXEC 159, we set up a transfer around the procedure declarations if this is necessary (corresponding to TRA θ above). Control in the productions is now transferred to PRI (which stands for procedure identifier). Upon entry to PRI an additional character has been scanned. Here we pick up the procedure identifier and change it to P-ID in the stack.

```
PRI    PROC    I     | ->    P-ID  |   EXEC 160   FND
FND    TYPE  P-ID    | ->    P-ID  |   EXEC 161   PSB
                <SG> | -»          | PSB EXEC 162 *(FPL
```

One sees from this subsystem of productions that EXEC 160 gets executed for all procedures, that EXEC's 161 and 162 get executed for functions, but that only EXEC 162 is executed for pure procedures since pure procedures are not preceded by types. In fact, EXEC 160 does everything common to procedures and to blocks. What we see, therefore, is that a division of labor is made between the several EXEC's handling these declarations so that labor common to several different compilation requirements is performed by a single routine. This organizational principle is found throughout the compiler. We have seen it before in the productions in the case of the production subroutine to process identifier lists. The structure of EXEC 260 is as follows:

```
RIGHT2 <-RIGHT3   CXT ;             (where CXT is current context)

CXT <-CODELOC;

<CXT> <-0 ; TALLY[CODELOC]:         (zero out context if procedure
                                     hasn't been called)
<CODELOO <~LEV + INC ;

   (here we won't know the block level nor will we know the
    increment [INC] until the end of the procedure declara-
    tion so a chaining mechanism is required.  Here we have
    oversimplified the presentation.)

LEV <^-LEV + 8R1000000;   (increments level count)

T <- FUNCTION ;  (sets up type for later entry into symbol table)

RIGHT1 «- LEFT1 ; SET[LEFT1, FUNCTION];

   (LEFT1 had the procedure identifier saved in it.  We
    transfer this description to RIGHT1,set the descrip-
    tion of LEFT1 to type FUNCTION, and push this de-
    scription onto the LADLE stack).

PUSH[LADLE,LEFT1]; PUSH[LADLE,CXT];   (we also push onto LADLE
```

the address of the first word in code where the contest will be stored. This corresponds to α in the code sample on page 35.)

PUSH[LADLE,0]; (finally, we put 0 on top of LADLE to delimit the code for the procedure body which ensues.)

We are now ready to do EXEC 161 for functions only and EXEC 162 for both functions and pure procedures. EXEC 161 says this:

F ← STORLOC ; (Save the head of the storage block in F)

TALLY[STORLOC]; (Save a word where value of procedure will be stored)

TYPE = DOUBLE → TALLY[STORLOC]; (If it was a real procedure save two words for a double precision result.)

T ← TYPE + PRCEDR ; (Save type and title of procedure for later entry into symbol table.)

EXEC 162 does the following:

ENTER[SYMB; RIGHT1, T, F, CXT]; (Here we enter into the symbol table the postfix identifier for the procedure, a type T set to function or procedure, a relative address F of the storage block for that procedure, and an address CXT where the run-time dynamic context will be located (this being α))

PUSH [STAB, 8L2+LOC[SYMB]]; SOnow in the stack STAB there are two words STORLOC where storage was interrupted and made relative, and the 2 flagged location in the symbol table where the procedure was stored causing that interruption of normal storage allocation.)

At this point in the productions we are about to scan the formal parameter list. Control in the productions is transferred to FPL where the following productions are encountered:

```
FPL     (    | →              | EXEC 157
                                SUBR SID      PCC
                                   ↑
                                 (identifier list subroutine entered)
  P-ID  ;    |                | EXEC 163      *S1
PCC     )    | →              |               *CCA
                                              (to treat parameter
                                               comment convention)
CCA     ( ;  | →              |               *VAL (look for value list)
```

EXEC 163 does nothing of significance to this discussion. It treats the case of parameterless procedures. EXEC 157 is entered before processing a formal parameter list to set things up properly. It looks as follows:

FNO ← 2; (Initialize count of formal parameter list to 2. The reason it is 2 is so that the integer can be used to access the thunk for that formal parameter by subtracting it from the mark [see code sample of procedure call on page 39 to understand this])

LOC[FPT] ← FPTLOC; (reset table for formal parameters to initial positions. FPTLOC initialized in EXEC0)

XEQ 190 ← FLST ; (Set up EXEC 190 [see pages 4 and 5] to execute the FSL code beginning at the label FLST)

Here FLST has code which looks as follows, and which is executed upon processing each formal parameter in the LEFT1 position:

'FLST' ENTER[FPT; LEFT1,FNO, FALSE] ; (Thus the postfix integer for the formal parameter, an integer used to access its thunk from the mark of the procedure call, and the Boolean value false are entered into the formal parameter table. The Boolean false will be set true for all formal parameters called by value as we will see soon.)

FNO ← FNO + 1 ; (here we tally the formal parameter number)

Next in the productions we expect to encounter the VALUE specifier telling us which, if any, of the formal parameters are to be called by value. This occurs in the productions at the label VAL. Before considering what happens at VAL we pause briefly to look at an example and to show what is built up so far.

```
REAL PROCEDURE P(A,B) ; VALUE A; REAL A,B ;
  IF A < 0 THEN P ← B+1 ELSE P ← P(A-1,B+3);
```

Up until the processing of the value list the FPT table for formal parameters looks like this:

A	2	FALSE
B	3	FALSE

After the processing of the value list the FPT table for formal parameters looks like this:

A	2	FALSE
B	3	FALSE

We see, therefore, that the processing of the value list consists of marking a TRUE in the third column of the formal parameter table opposite the formal parameter in column 1. The following productions and exec routines accomplish this.

```
        VAL        VALUE  |              | EXEC 172
                                           SUPR SID   VLU
```

EXEC 172 does XEQ 190 ← VLST; to set up EXEC 190 to process the identifier list as a value list, whence for each identifier on the value list we do

```
        'VLST'     FPT[LEFT1,,$] ← TRUE ;
                   ¬SIGNAL → FAULT 5 $
```

At VLU in the productions we expect to have finished processing the value lists and we turn to the specifier lists:

```
        VLU       VALUE ; | →        |          *SP ←(for specifiers)
                     <SG> |          |        ERROR
        SP           <SG> |          |      SUBR CHG    SPA
        SPA          TYPE |          |      EXEC 167    *SP2
        SP2             I |          |    ISP SUBR ID   SPT
```

[more productions are inserted here to treat other kinds of specifiers like <u>array</u>, <u>procedure</u>, <u>label</u>, etc. We will discuss only one case.]

In EXEC 167 we set up EXEC 190 to process specifier lists.

```
                   XEQ 190 ← SLST ;
```

The code at SLST being as follows:

```
        'SLST'    FNO ← FPT[LEFT1, $ , ] ; (retrieve formal parameter
                                             number from table)
                  ¬SIGNAL → FAULT 6 ; (if don't find it in table then error)
                  FPT[ 0,,$] → (Here if was true then had call by value,
                               so write code to compute formal parameter
                               by value and to store it away as follows)
```

```
T ← ABVAR; (set up type for later table entry)

MARKJUMP[ DECLARE ]; (

CODE(MARKJUMP[V203]);

<CODELOC> ← (THUNK +FNO)xSHIFT +CXT;

    (here we code a word with the appropriate
     thunk code [see page 40], 005 in this case,
     plus the formal parameter number and the
     address in code where context is located =
     005 α2)

TALLY[CODELOC];

LEFT4 ← LEFT2 ;

RIGHT2 ← TYPE + RZ;  (Where RZ is a storage constant)

JUMP [STORE];  (here STORE compiles code to store
                the formal parameter called by
                value whose value has just been
                computed by V203.)
```

The code produced by this call by value process looks as follows:

```
α CONTEXT WORD

  LEV    INC

  TRM V203  }
            } - - ▸ Compute value of first formal parameter
  005 α,2   }

  CLA 3  R0  }
             } - - ▸ Get value from standard location
  STD 3  /77 }       where left by V203 and store
                     indirectly, /77 giving local
                     context.
```

We now return to the code for SLST. For formal parameters not called by value we have:

```
ENTER [ SYMB; LEFT1, TYPE+THUNK,FNO, CXT ];
```

Thus, information about the processing of formal parameters has been entered in the symbol table so that upon encountering the formal parameters in the body of the procedure the correct accesses are compiled to the thunks in the call of the procedure. The productions determine the scope of the body of the procedure and techniques are used to remove the formal parameters from view in the symbol table upon completion of the processing of the procedure body. These techniques involve opaquing certain entries in the table by scatter repeat chaining.

Let us now take a look at what happens at the end of a procedure. After scanning a statement all characters in that statement are eliminated and control is passed to production subroutine E30 after 'END' or ',' has been scanned. E30 determines whether or not a procedure declaration is being terminated by means of a production of the following form:

```
E30    PROC ├  ;  | →        | EXEC 35    *CNT
```

and at CNT we see

```
CNT    <DC>  |           |  DEC
       <SG>  |           | EXEC 165  RETURN
```

Hence EXEC 35 is executed once after each procedure declaration and EXEC 165 is executed once at the end of all procedure declarations. Here EXEC 35 looks like:

```
MARKJUMP[SASS];  (←which assigns storage requirements)

ENTER[SYMB ; STAB, 0, 0, 0] ( ← this opaques the portion of the
                               symbol table containing formal
                               parameters for the recent procedure
                               by inserting a 2 flagged address
                               which jumps to a previous portion
                               under a scatter repeat search test)

POP[STAB,STORLOC]; (←this returns STORLOC to previous value before
                    it was set to contain relative addresses)

CODE ( JUMP [ V202] ) (←where V202 returns to call, c.f. p46)

MARKJUMP [ATLAS] ; (←assigns chains, c.f. pp44-45.)

CXT ← RIGHT1 ; (restore context saved by EXEC 160)
               (was saved in semantic stack under left terminator)

CLUTCH ← TRUE ; (set switch to denote that this code cannot be
                 gotten to by the flow of control of compiled
```

code, i.e. control can come only via transfers from the run-time routines for procedure administration)

LEV ← LEV - 1 ; (decrement static block level)

Upon exit we see that CXT contains the address of the head of the code generated upon entrance to the procedure declaration just processed. EXEC 165 , now, says the following:

CLUTCH → ASSIGN[FLAD4] ; CLUTCH ← FALSE;

this merely assigns the transfer coded around the batch of declarations produced. It corresponds to the command TRA θ in the code sample on page 46.

Let us now take a look at the code produced corresponding to the formal parameters found in the procedure body. Recall that all formal parameters have been entered in the symbol table after the processing of the formal parameter list and after the processing of the specifiers. Corresponding to each formal parameter is a line in the symbol table which has in it POSTFIX INTEGER, TYPE + THUNK, FNO, CXT (←cf.p53).

The EXEC responsible for producing accesses to variables which do not occur on the left hand side of assignment statements is EXEC 7. It is called by the following subsystem of the productions in the expression scanner.

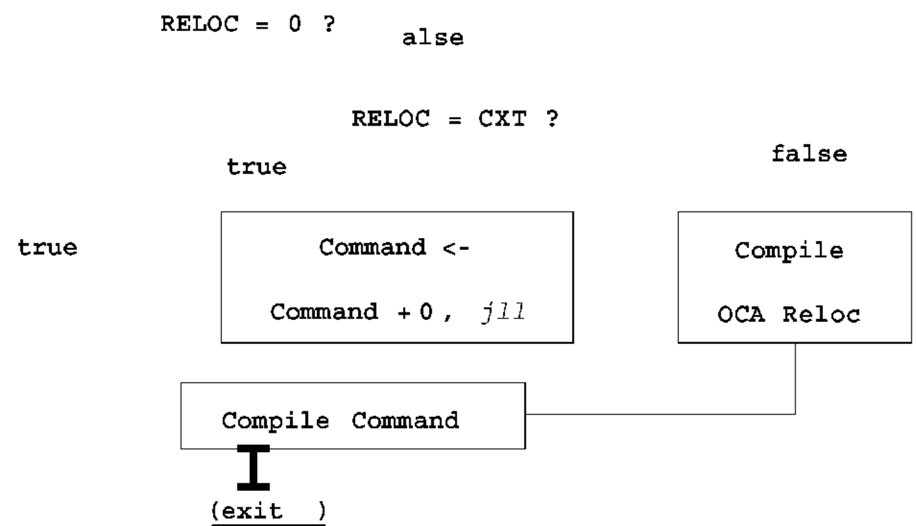
```
E1      I | →        E |            *E2
E2    E ( |
      E ← |
     .......
     E <SG>|            |     EXEC 7
```

and so we see that EXEC 7 is called only in the event that we have a simple identifier not followed by a storage operator, ←, a right bracket, (or [, or a comma. EXEC 7 reads the information about the identifier in the symbol table and analyzes what code to produce (to access that variable). EXEC 7 calls the semantic subroutine FIND which looks up the identifier in the symbol table, puts its class in the accumulator, its relocation base in RELOC, its relative address in KEY, and its type in TYPE. It then returns to EXEC 7 where its class is placed in the OA register and used to select a transfer command in a switching table, which switching table transfers to different routines to process the different kinds of variables classified. Let's take the case of an integer variable. EXEC 7 sets up information in the semantic stack and in a special stack called BASE, which stack has one entry for each expression E in the syntax stack. In the semantic stack corresponding to the integer variable it puts RIGHT2 ← KEY + MODE + TYPE + TEMP to set the types and addresses for the MAGIC compiler. Here, KEY gives the relative address, MODE gives the mode of the access to the variables, TYPE gives the type of the variable, and TEMP has a bit in it specifying whether or not the variable is relocatable or fixed. These items make up the description of the variable. A further statement BASE ← RELOC puts the current relocation base [0 outside of all procedures, and non-zero inside procedures] in the BASE stack. The code compiled for accessing integer variables will then be the following for the following three cases:

(1)	CLA KEY	if RELOC = 0 and we are outside all procedures
(2)	CLA KEY, /77	for variables where RELOC = current local context, the local context coming from /77
(3)	OCA RELOC CLA 2 KEY	for variables where RELOC ≠ current local context.

A flow chart expressing the discrimination between these three cases is found on the top of the next page.

Assume Command has in it a command you want to compile immediately.



To see this in more detail let's consider a specific example.

Suppose we want to compile code for a program with a structure as follows:

```
BEGIN REAL A
20200____7>BEGIN PROCEDURE X
               r BEGIN  INTEGER  B
21100______________> PROCEDURE Y
        <       (       CBEGIN HALF  C
                                ... B + AxC
                         END
                ^ END
          END
END
```

When compiling the expression B + AxC in the innermost block the syntax stack will, at some point, contain E + E x E. By the time this is built up entries for all of the identifiers have been made in the symbol table as follows:

ID	TYPE + CLASS		KEY	CONTEXT
A	REAL	VARIABLE	40000	0
B	INTE	VARIABLE	1	20200
C	HALF	VARIABLE	3	21100

Furthermore, EXEC 7 will have inserted descriptions in the semantic stack corresponding to each variable, and it will have built up the BASE stack with relocation bases. The picture of these various stacks is as follows:

E	+	E	X	E	← syntax stack
inte-ger mode 1		real 40000		half 3	← semantic stack
20200		0		21100	← BASE stack

The routines to compile code for arithmetic operations, which are the EXEC's in subroutine COM, have the capability of analyzing the information in the semantic and BASE stacks and of being able to produce the correct code. This code will look as follows:

```
CLA   40000    ACC ← A
MPY   3,77     ACC ← AxC
OCA   20200    ACC ← ACC + B
ADD 2   1
```

Notice that this example uses all three cases discussed on the bottom of page 43.

BLOCK ADMINISTRATION

There are two cases that must be considered. The first is the case when blocks are outside of procedures. In this case we push the STORLOC onto a stack at entrance to a block and reset it upon exit from the block. The stacking mechanisms allows us to handle nested blocks. The second case is when blocks are internal to procedures. Here block administration must be set up to handle recursion. The mechanism must be set up to store in the code itself the storage requirements for a given block. Of necessity, things become more complicated. Let us try to get an understanding of the problem first by considering the example below.

action	program	storage required	STORLOC
size=chain[L1]	PROCEDURE P(M,L); VALUE M; REAL L,M;	4	5
size=chain[L2]-chain[L1]	BEGIN REAL A; INTEGER B;	3	5 → 8
size=chain[L3]-chain[L2]	BEGIN REAL A; INTEGER C;	3	8 → 11
assign L3-storloc	END		11 → 8
	A ← A×B - A×2;	2 temps	8 → 10
size=chain[L4]-chain[L2]	BEGIN FORM X, Y;	4	10 → 14
size=chain[L5]-chain[L4]	BEGIN FORM Z,G,X;	6	14 → 20
assign L5=storloc	END		20 → 14
	X ← Y×3 + (A-B)×(Z×G);	2 temps	14 → 16
assign L4=storloc	END		16 → 10
	B ← (A-B)×(A+B);	2 temps	10 → 12
assign L2=storloc	END		12 → 5
assign L1=storloc	;		5

In concise and abbreviated form what we are going to do is this. We will keep STORLOC in a stack at the entrance to each block, and we will reset it to the value saved upon exit from that block. We augment STORLOC whenever we hit declarations which require storage or whenever we require temps to compute an expression within a block. The storage required for a block is, therefore, computed by subtracting from the value of STORLOC at the instant of exit from the block, the value of STORLOC at the instant of exit from the block of level one lower in which the given block is imbedded. Since these quantities are not known at entrance to each block, a chaining mechanism must be set up to compute them. The storage requirement of the procedure in which all of these blocks are imbedded is the value of STORLOC upon exit from the procedure.

To see this more clearly, let's take a look at block 2 in the example on page 46. Before entering block 2 the value of STORLOC is 5. When we enter, three cells are needed for the declaration REAL A; INTEGER B;. This augments STORLOC to 8. Then we hit the imbedded block 3 which increments STORLOC to 11 for its own storage requirements, but which resets it to 8 upon exit, thus having no incremental effect on the STORLOC counter for block 2. Next, we hit an expression which is in block 2, and which requires 2 temps, and we see that STORLOC is incremented to 10. Processing block 4 and its imbedded block 5 have no effect on STORLOC for block 2, since STORLOC is reset to the same value upon exit from block 4 that it had upon entrace to block 4; namely, it is reset to 10. After processing block 4 we process another statement in block 2 requiring temps,and this increments STORLOC to 12. The value 12 is thus the value of STORLOC upon exit from block 2. The inner blocks in block 2 have had no incremental effect on this value of STORLOC by the time we exit block 2. The total storage requirements for block 2 can thus be determined by subtracting from 12 the value STORLOC will have upon exit from the procedure [i.e. the block in which 2 is imbedded, which has level one less than that of block 2]. The resulting difference is the difference between the storage reserved for the procedure and the storage required for block 2. This difference is the increment to storage which must be reserved at run-time every time the run-time flow of control leads us to enter block 2, be it recursively or otherwise. The increment is thus stored in the code in order to be processed by the run-time routines that handle dynamic storage allocation. Thus, we see that 12 - 5 gives 7 words required for block 2, so the number 7 is stored in the code near the entrance to block 2, and 7 additional words of dynamic memory space will be reserved at run-time every time we enter block 2. Let us now take a look at block 3 embedded in block 2. We see that three words will be required for block 3, but that among the seven words reserved upon entrance to block 2, four are needed for expressions which are evaluated after leaving block 3. Thus, the storage requirements for block 3 are overlapped on the storage requirements saved by block 2. This means that no words are required for block 3. We see that by subtracting the value of STORLOC upon exit of block 2 from the value of STORLOC upon exit of block 3 we get 11 - 12, or -1. Thus, our algorithm can conclude that enough storage is reserved for block 2 to completely suffice for the requirements of block 3 and no storage need be reserved for block 3. In a similar fashion, we see that four words of storage are required for block 4, and that 4 words of storage are required for block 5. If the reader has understood thus far the problem and the fundamental method of determining the storage requirements for blocks inside procedures he will be prepared to understand the following algorithm in FSL used to implement the solution by means of chaining.

The FSL solution is as follows. For each procedure and for each block we reserve one word in code with a left half and a right half

LH	RH

.

LH points to the next block word on the chain of block words unless it is zero (which indicates the end of the chain).

RH before end of block, points to chain of inner block words, and after end of block, indicates value of STORLOC at end of block.

We further have the following table of cells relevant to the semantic routines.

CSS is a cell pointing to the current block size word.

LSS is a stack containing previous block size word locations (which stack is used as backward links on the chain of block size words, enabling us to back up on the chain).

CODSTK is CODELOC except it is of type LOGIC.

X is the address extractor 8R77777.

SHIFT is left shift 15 bits, 8R100000.

R15 is right shift 15 bits, $8F1_{10}$-5.

LEV is the current block level required in proc. size word.

X85 is the block entry routine.

LXPRO is the opcode and index register required on the final command.

We now have four semantic routines to accomplish the chaining:

↓procedure entry↓

```
PUSH[LSS,CSS]; CSS ← CODELOC;
CODSTK ← LEV; TALLY[CODELOC];
```

(here we put the previous current storage setter, pointing to previous block size word on the chain of reverse links, LSS, set CSS to CODELOC obtaining a new block size word, save the static level in CODSTK and tally CODELOC)

↓block entry↓

```
CXT → CODSTK ← (CSS>∧x7) × SHIFT;
<CSS> ← (<CSS> ∧ ¬ X7) + CODELOC;
PUSH[LSS,CSS]; CSS ← CODELOC;
CODE (MARKJUMP [X85]);
```

(here if CXT is non-zero we are inside a procedure, and we execute the ensuing statements inside procedures only. We then extract the address from the previous value of the current storage setter, shift it left 15 and store it in CODSTK. Then we chain the right half of the last block size word to the present codelocation. This present codelocation becomes the new block size word, and we push CSS onto LSS and reset it to CODELOC.)

↓block exit↓

```
CXT → MARKJUMP[SASS] $
```

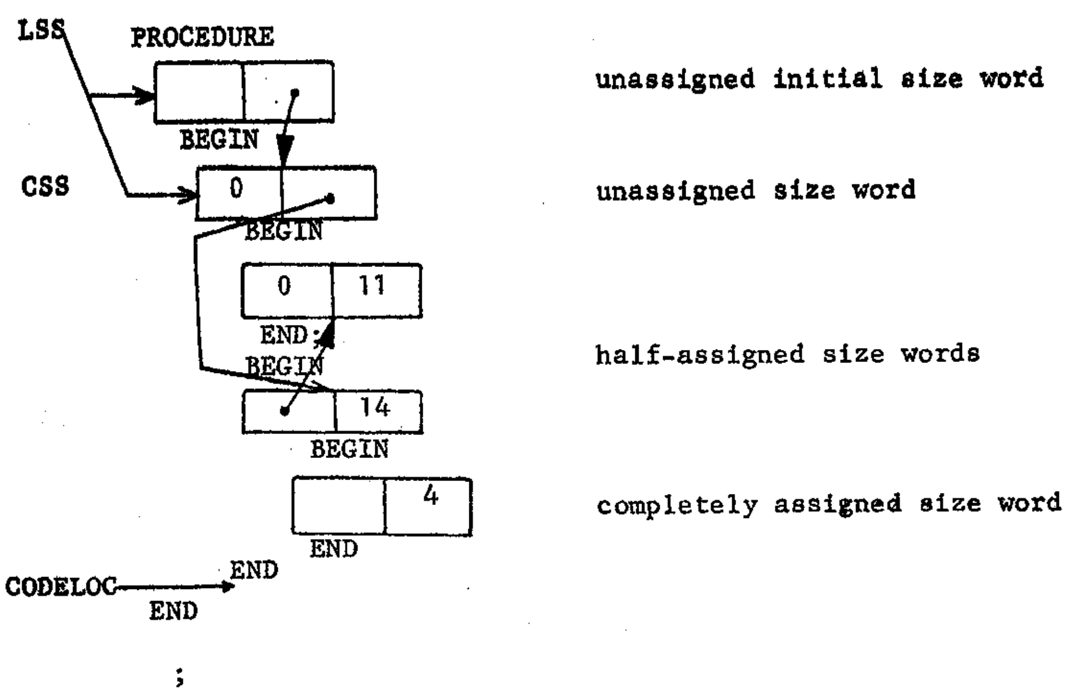
(here if we are inside a procedure we markjump to SASS).

↓procedure exit↓

```
          MARKJUMP[SASS];
"SASS"    T ← <CSS>∧X7; <CSS> ← (<CSS>∧¬X7) + STORLOC;
'SAS'     T → TT ← <T>×R15; <T> ← (<T>∧X7) - STORLOC;
          T ← TT;  JUMP[SAS] $; POP[LSS[CSS]; JUMP[<SASS>];
```

(As is seen this routine is shared by procedure endings and by block endings for blocks inside procedures. First we save the address portion of CSS in T. Then we replace the contents of CSS with the same left half and assign the right half the current value of STORLOC. If the right half was non-zero, then we are not at the end of the chain of inner blocks (the right half having been stored in T, which is tested for a non-zero status) and the previous right half pointed to the next block size word on the chain of inner blocks. Thus, we shift the address of this next block size word to the right 15 places and store it in TT. Then we subtract the current STORLOC from the previous STORLOC stored in the right half of the block size word which right half contained the value of STORLOC upon exit from that inner block. This difference is the storage requirement [a line should be inserted above at this point to set this storage requirement to zero if the difference is negative]. Finally, we place the contents of TT in T and iterate the cycle at SAS to compute all of the differences on the chain of inner blocks and to assign them as storage increments in the block size words. If, on the other hand, T was, or becomes, zero at any stage of the loop SAS, we pop LSS onto CSS to return to an outer block one level up in which the current block is embedded. Then we leave SASS. Thus, the stack LSS contains the reverse of the history of descent into blocks,and it allows us to ascend back out when inner blocks become processed.)

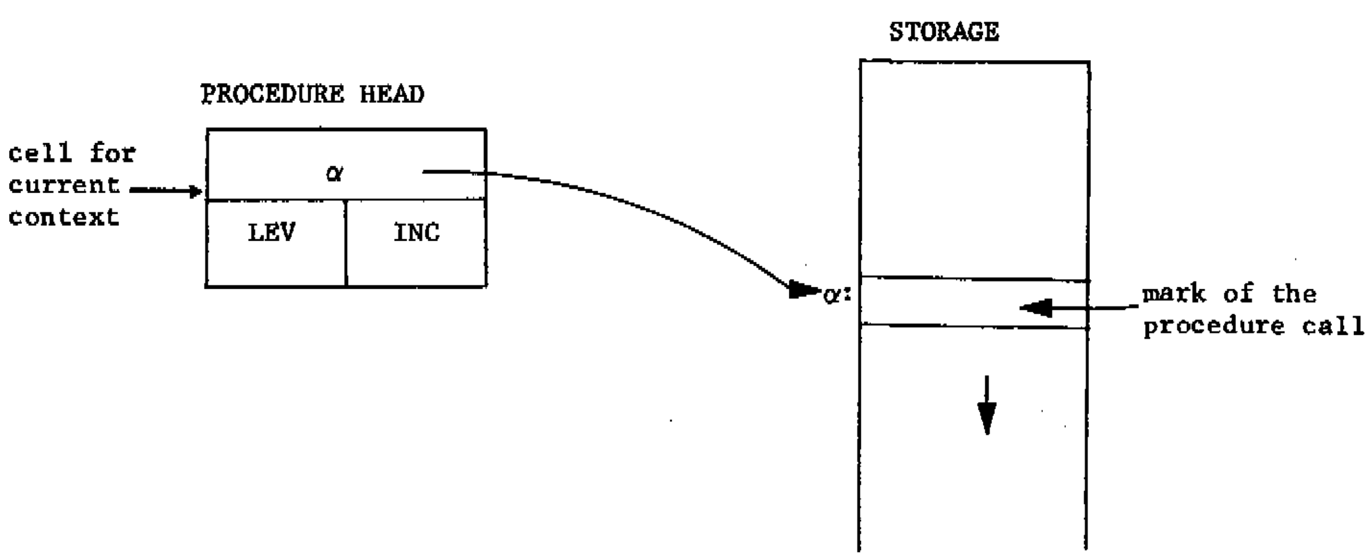
The reader is advised to work through an example of this chaining mechanism to get a really clear understanding of it. To help,a diagram is provided following below, with different dotted lines showing various stages of evolution in the chaining process.



This example shows the state of the storage size chains at the point in the compilation when CODELOC is as indicated. All possible variations of the storage size words are represented in this example. We see that CSS is pointing to the current block size word. Further, LSS, the stack containing the history of descent into the block structure, is pointing to the procedure head. Each block size word must be assigned twice. The comments on the right indicate each of the four possible states of assignment. As is seen, the right hand linkages point to the last block within the current block, and the left hand linkages point to previous block at the same level. (This last statement is general.)

RUN-TIME RECURSION ROUTINES

There are two stacks used at run-time to administer storage allocation, the STORAGE stack itself, and the HISTORIAN, which, among other things, keeps a trace of procedure calls. The current context cell in the head of a procedure will point to a location in STORAGE which is the current base of storage for the most current call on the procedure.



One resets storage on the way out of procedures by using information stored in the historian. When one enters a procedure, one stacks a word pair on the HISTORIAN which contains [procedure name, address of first word of code for procedure] = first word, and [previous storage pointer for that procedure] = second word. When one enters a block one stacks a single word on the HISTORIAN containing [static level, beginning of dynamic storage for that block]. A third possibility in addition to procedure entries and block entries is a parameter call entry. Here the HISTORIAN is manipulated to simulate the state of the call where the formal parameter is to be computed. The manipulation consists of inserting a marker in the stack, of copying certain information and of putting a two-flagged link in the stack which opaques part of it to scatter repeat searches causing the re* suit to simulate the proper state of the machine for the formal parameter call. Later, tine two-flagged link is removed, and the previous state restored. On the way out of procedures and blocks storage is reset using information stored in the HISTORIAN.

To see with clarity what is going on we need to consider an example* Suppose with the call statement we call procedure **P**(X) where X is a formal parameter **P**(Y+Z). Suppose further that within the declaration **P**(X) there is a call on R, and that within that call on R there can occur another call oh R followed by a use of the formal parameter X. Then suppose that at run-time this calling pattern happens. When **P**(Y+Z) is called the HISTORIAN is augmented to look like **P** * where **P** is the location of the procedure head in code, and where p is the previous storage pointer for the most recent use of **P.** Upon procedure entry the context of **P** is set to the current top of STORAGE, and the current top is incremented by the INCRMENT to storage required by the procedure (which increment is stored in the head of the procedure at compile time). Upon entering R the HISTORIAN is changed to look like «_ **R** R, - **P** . p *i *• the previous storage pointer corresponding to the most recent call of R. Upon entering R the second time (within itself) the HISTORIAN is changed to look like $\leftarrow\boxed{R}\boxed{r_2}\boxed{R}\boxed{r_1}\boxed{P}\boxed{p}$ where r_2 is storage pointer used for the call of R just mentioned. Now we must compute the value of the actual parameter Y+Z corresponding to its use in place of the formal parameter X. The object code gives us the thunk number, and the procedure call location corresponding to the actual parameter Y+Z. But to execute this thunk we must return to the state of STORAGE that prevailed at the entry to P. But before returning we must make provision to restore the HISTORIAN to the present state. Suppose the current context of P is p' and that that of R is r' and that the location in code where we are calling X is t. Then we put -t in the HISTORIAN as a boundary marker, and we stack $\boxed{R \;\; r'}$ and $\boxed{P \;\; p'}$ on top while changing the contexts of R and P to r_1 and p, respectively. The HISTORIAN now looks like this

←	P	p'	R	r'	-t	R	r_2	R	r_1	P	p

with the current contexts of R and P set to r_1 and p. We finally stack a 2-flagged link around this entire stack to make it look like

←	$\overset{2}{\text{flag}}$	P	p'	R	r'	-t	R	r_2	R	r_1	P	p	

At this point the HISTORIAN looks exactly like it did at the point before entering P, and we now compute the thunk for the formal parameter and deliver the address of the value. Thus, we see that the environment in STORAGE where the actual parameter is computed is identical to the environment outside of the procedure call [as it should be in the definition of ALGOL 60. Consider X + P(X)]. Now, having computed the value of the actual parameter we must restore the environment in STORAGE that existed prior to computing the actual parameter. This means popping the HISTORIAN back to the marker -t, resetting contexts as we go to p' for P and r' for R. Everything back to and including -t is popped off. Thus, the proper environment is restored, and we continue executing object code at the address t. Within the procedures P and R we could have crossed block boundaries resulting in the stacking on the HISTORIAN of block storage pointers, and in the removal of such pointers. The above manipulations of the HISTORIAN are not altered by the stacking of block storage pointers since the search processes ignore them. When one leaves a block or a procedure by a normal exit (i.e. by going across the begin-end boundary rather than by leaving by means of a designational expression) one resets STORAGE (in the case of blocks) or resets the context (in the case of procedures) to its previous value by means of the most current entry in the HISTORIAN corresponding to the block or procedure. Exits by means of designational expressions are accomplished by storing destination address and destination level in the code and by transferring to a run-time routine which pops the HISTORIAN until it finds the proper target level (level information being stored in the HISTORIAN along with each entry). Notice that for formal parameters which can be designational expressions and for actual parameters which contain function calls where the result of the call is a go to, the opaquing feature constructed in the HISTORIAN during the process of actual parameter evaluation will result in a proper search for the target level during the execution at run-time of a designational expression. [This is a pretty hard thing to notice without working through an example. The reader is advised to do this.]

FORMULA MANIPULATION

DATA STRUCTURES FOR FORMULAS

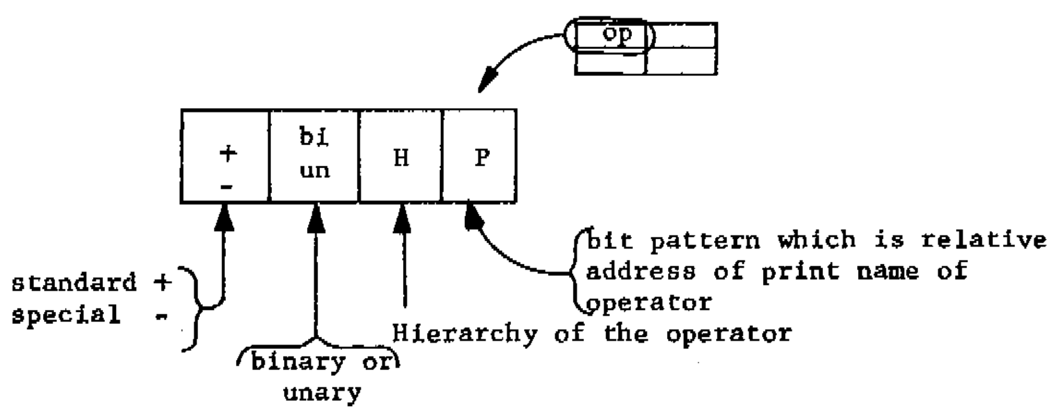
There are two kinds of formulas, standard and special. The standard formulas comprise those made from binary or from unary operators with two or one operands respectively. These are constructed from word pairs taken from the list of available space, and linked together. For binary operators the building block looks like

α	binary operator	operand A	word pair from available space
$\alpha+1$		operand B	

For unary operators the building block looks like

α	unary operator	operand A	word pair from available space
$\alpha+1$		don't care	

The operator portion of each word pair contains the following information:



The operands A and B consist of a <u>tag</u> and an <u>address</u>:

TAG	ADDRESS

The tag is a bit pattern giving the type of the object referred to by the address. These types include integer, floating point number, formula, text, chain, logic, and atomic formula. For an integer tag the address points to a word containing the integer if the integer is greater than 15 bits, otherwise the integer is stored as the address. For a floating point number the address points to a word pair containing the number in double precision form. For a formula the address is the address of the head of the formula. For the text tag the address is the relative address of the print name of the text. For the chain tag the address is the address of the head of the chain. For a logic tag the address is the address of the logic word. Finally, for the atomic formula tag the address is the relative address of the print name of the atom. The routines to construct formulas from these building blocks are fairly straight forward. They take their operands in a fixed locations, such as the accumulator and various index registers, and they construct the formula using word pairs taken from available space by setting up the operands and operators of the building blocks so that they contain the proper information and link to the proper successors.

The special formulas correspond to the source language constructs .ARRAY, .PROCEDURE, . ← and |op|. These correspond to data structures using chains as operands. Chains will be explained later in the list processing section. Suffice it to say, for the present, that parameter lists for postponed array accesses or for postponed procedure calls are stored as chains.

OPERATIONS ON FORMULAS

The syntax of formula manipulation is straightforward and not worth commenting on in detail. For an understanding of the syntax of formula manipulation the reader may look at the syntax listing. He should have built up enough feeling for the system by this point to understand the syntax of formula manipulation without difficulty. The semantics is also relatively straight forward and the same remarks apply.

The crucial powers of formula manipulation lie in the run-time routines. This is the case because most actions involving formulas are either interpretive at run-time or involve manipulations which cannot be compiled into the object code as macros because of the size of the code involved. We shall examine here four main run-time routines communicating their actions by means of flow charts. These four routines lie at the heart of the run-time system. The reader will recall that one crucial mechanism used in handling recursion for the run-time routines was discussed on pages 15 and 16. The use of this mechanism will be implicit in the flow charts discussed.

The Print Routine

The print routine is discussed because it involves a switching mechanism found ubiquitously in the run-time routines for formula manipulation. Upon entry to the routine an operand, consisting of a tag + an address, is found in the accumulator. One executes a mark transfer to V6 which routine saves the address portion of the accumulator, analyzes the tag, and provides a return jump to the mark plus the tag. This provides a rapid discrimination on tags, each tag producing a jump to a separate portion of the run-time code for processing.

TRM	V6	save address and come back with jump to appropriate entry point
LWD	E1	entry point for integer printing
LWD	E2	entry point for f.p. number printing
LWD	E3	entry point for formula printing (recursive)
LWD	E4	entry point for text printing
LWD	E5	entry point for chain printing
LWD	E6	entry point for logic word printing
LWD	E7	entry point for atomic formula printing

The respective entry points are addresses in assembled code where the printing instructions for a given type of data are to be found. In the case of formula printing the code can call the entire routine recursively. The sequence of actions for this is:

E3 set up recursion, print operator if unary,
save second operand if operator binary, save operator if binary,
print first operand recursively, pop up, if had binary case
print binary operator, then print second operand recursively.

The Eval Routine

There are two cases in the syntax of the source language which call the evaluation routine. The first of these cases is transformed into an instance of the second.

I. $G \leftarrow \text{EVAL } (X_1, X_2, \ldots, X_n) \text{ F } (E_1, E_2, \ldots, E_m);$

II. $G \leftarrow \text{EVAL } ([T]) \text{ F } ([S]);$

where T is a chain of formal parameters and S a chain of actual parameters.

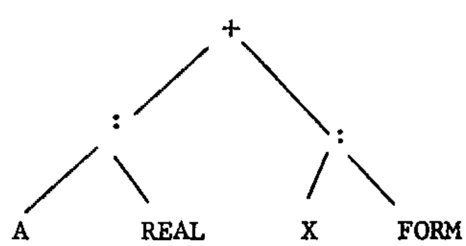
As far as the semantics are concerned we check the type of F, and if it is other than a formula we compile a normal assignment statement $G \leftarrow F$. For the first case above we compile code to construct the chains of formal parameters and actual parameters. The cells to construct these chains are taken from available space. They are discarded afterwards. For the second case the code produced will be:

```
CLA  T
STD  Y3
CLA  S
STD  Y4
CLA  F
TRM  EVAL
```

The flow chart for the eval routine is found on the next page. Notice that it performs simultaneous substitution of actual for formal parameters.

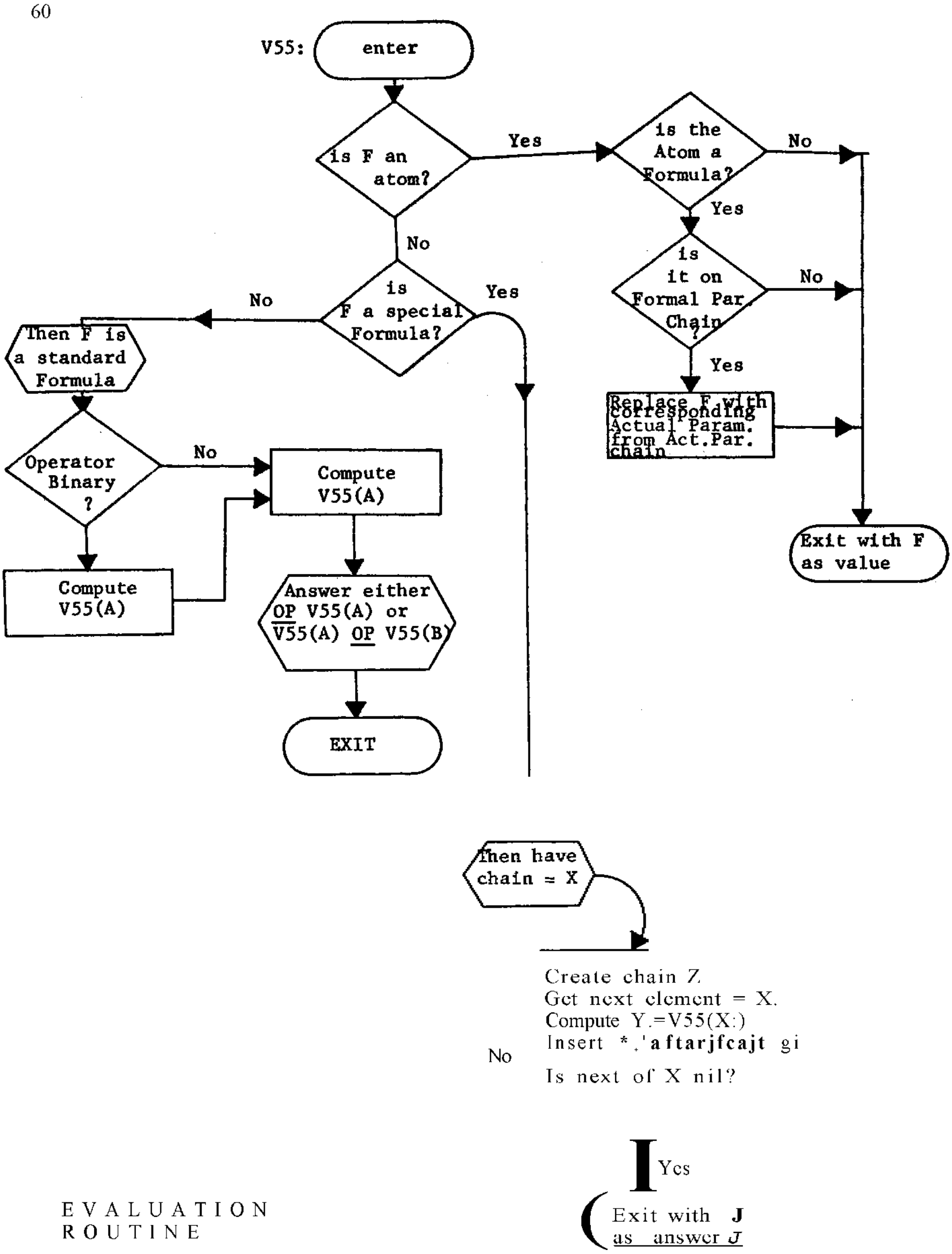
<u>The Pattern Routines</u>

Consider the expression $F == P$ where F is a formula, say $F \leftarrow 3.8 + A \times 2$, and where P is a pattern, say $P \leftarrow A{:}REAL + X : FORM$. The colons in the pattern P are treated as binary operators. Thus, P might be represented as:

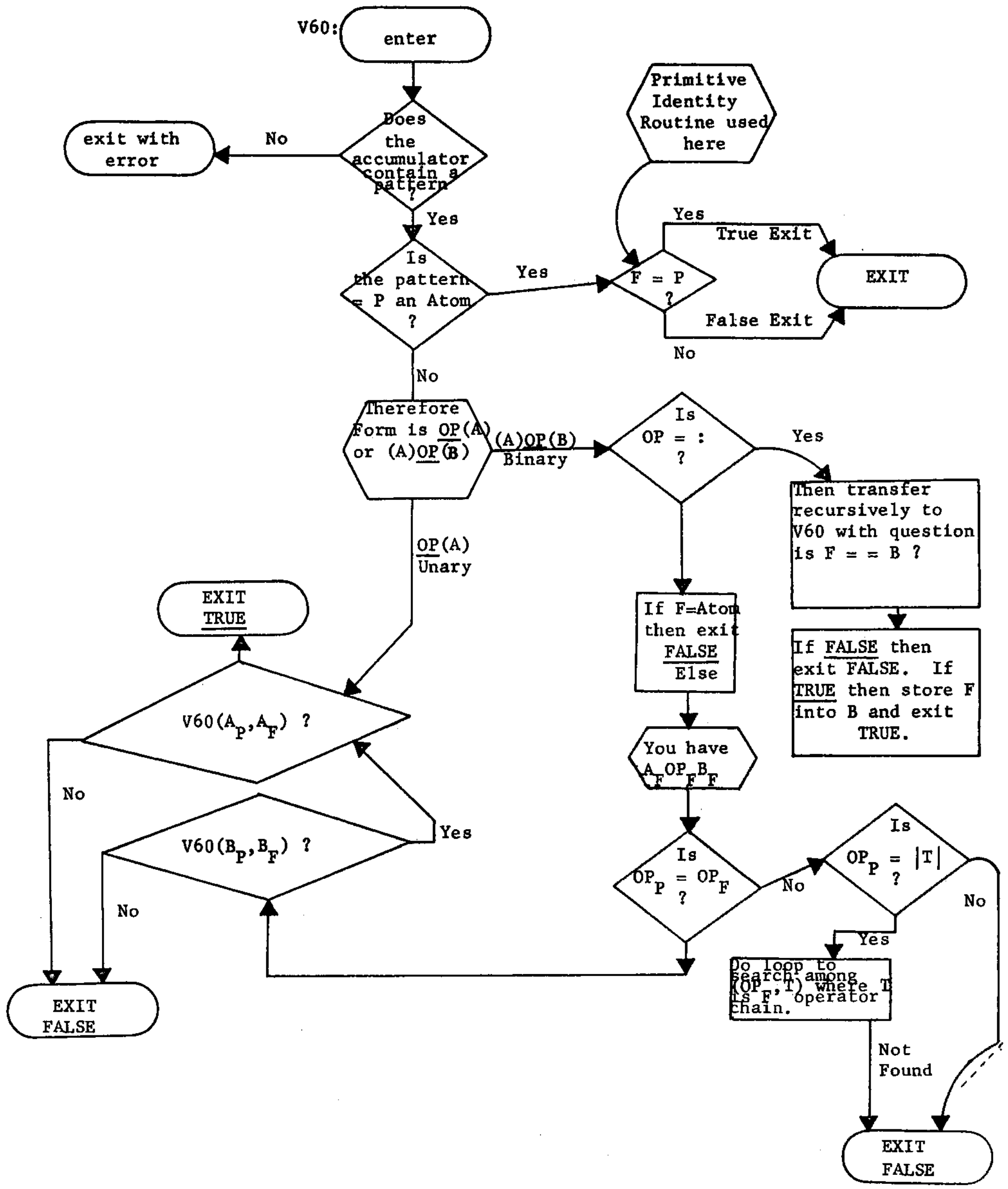


When it is determined that an operator in the pattern is binary, that operator is checked to see if it is the extractor operator ':'. If this is the case the left hand operand is saved, the test is performed on the right hand operand, and should the result of the test be true the formula (or subformula) of F matching the right hand operand of the pattern is assigned to be the contents of the variable which is the left hand operand of the extractor. The flow chart for the exact identity pattern routine V60 appears on page 61.

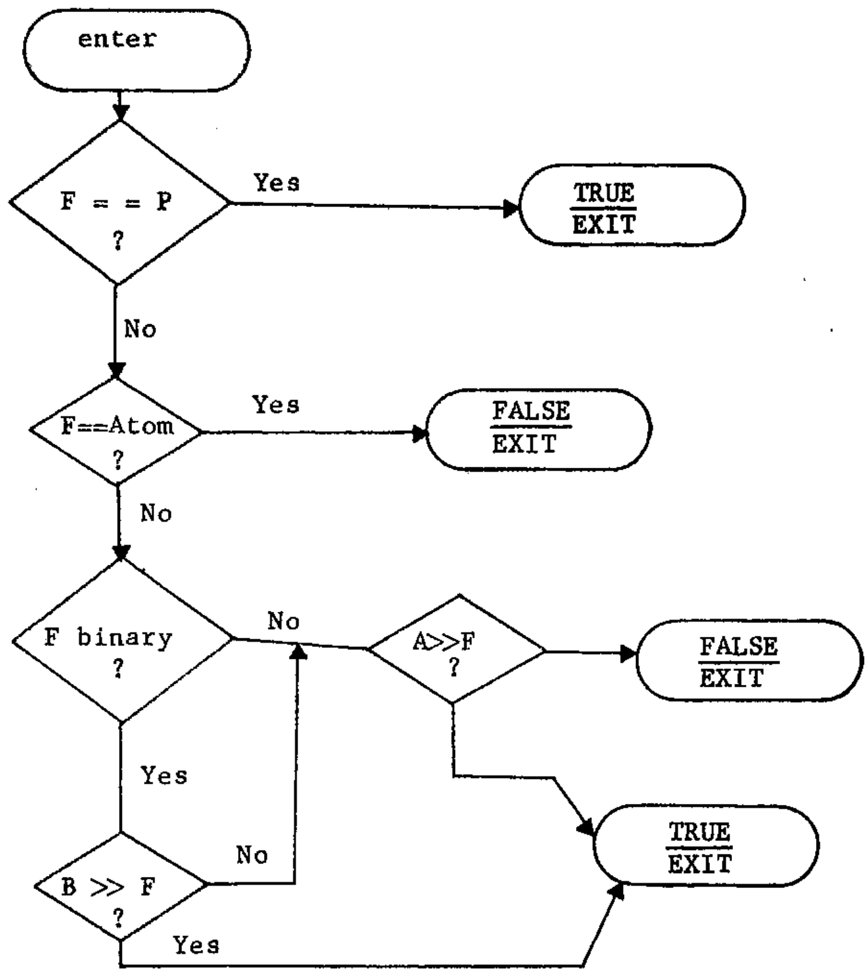
The flow chart for the routine to perform $F \gg P$ appears on page 62. Notice that it uses V60.



EVALUATION
ROUTINE


V60:
enter
Does the accumulator contain a pattern ?
No
exit with error
Yes
Is the pattern = P an Atom ?
Yes
Primitive Identity Routine used here
F = P ?
Yes
True Exit
EXIT
False Exit
No
No
Therefore Form is OP(A) or (A)OP(B)
(A)OP(B) Binary
Is OP = : ?
Yes
Then transfer recursively to V60 with question is F = = B ?
If FALSE then exit FALSE. If TRUE then store F into B and exit TRUE.
OP(A) Unary
If F=Atom then exit FALSE Else
You have A OP B
EXIT TRUE
V60(A_P,A_F) ?
No
V60(B_P,B_F) ?
Yes
No
Is OP_P = OP_F ?
No
Is OP_P = |T| ?
No
Yes
Do loop to search among (OP_F, T) where T is F, operator chain.
Not Found
EXIT FALSE
EXIT FALSE

F >> P:



The Interpreter

As our last topic in the treatment of formula manipulation we mention a very neat interpreter which is implemented using the XEQ instruction. For interpreting formulas with arithmetic operands of the form A op B we have a mapping taking the operator into an integer, which integer is stored in the index register R1. Then we do

```
CLA    A
XEQ    ZO,R1
```

Here ZO is the address of the head of a table of interpretive arithmetic commands:

```
ZO    ADD    B
      SUB    B
      MPY    B
      TRM    Exponents
      ....
```

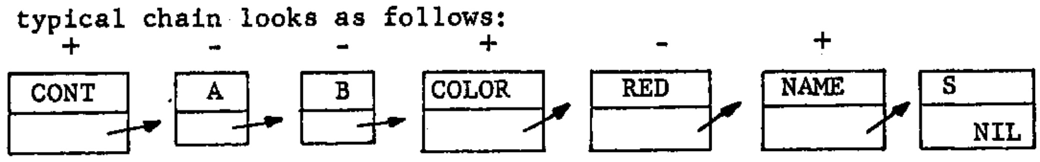
The command performed by XEQ is that located at ZO + the contents of RO. The integer in RO thus switches the XEQ to the proper operation.

LIST PROCESSING

DATA STRUCTURES FOR LISTS

The data structures for lists are sequences of word pairs, the second member of each pair containing a 2-flagged address to its successor pair in the sequence, and the last pair being linked to a special cell NIL. Pictorially this looks like:

The address α of the first word of the first pair in the chain is the address of the chain. Given this address we can scatter repeat down the chain searching for some property of the contents of the first word of each pair in the chain. If we further place in the cell NIL an object we are searching for, we are guaranteed to find it either on the chain or in the cell NIL. If we find it in the cell NIL this means it wasn't on the chain. Every chain is a description list containing a sequence of attributes and values. Each attribute is followed by a list of values associated with it. There are always two standard attributes on a chain, the contents attribute CONT, and the print name attribute NAME. The contents attribute is always the first on the chain, and the print name attribute is always last. Other arbitrary attributes are placed in intermediary positions in the chain by the system. If + stands for attribute and - for value, then a typical chain looks as follows:



The items stored in a chain as values may be any of the operands legal in a formula (c.f. pages 55 to 56) as an operand. These are called data terms and are so marked. In addition, we may store symbol variables and local chains. Each of these possibilities is stored in the first word of a pair on the chain. The second pair is reserved entirely for the link to the next pair or to NIL.

THE CHAIN ACCUMULATOR

At the heart of the list processing system lies a stack of word pairs called the chain accumulator. It holds pairs of pointers pointing to the right and left hand ends of chains or subchains. For example, the first pair on top of the chain accumulator in figure 3 below is (a_1,a_2). This is a pair of addresses pointing to the head and tail of a chain. Likewise with the pair (b_1,b_2). To concatenate these two chains we must link the tail of the second to the head of the first and fix up the chain accumulator. Figure 4 shows the result after concatenation has been performed.

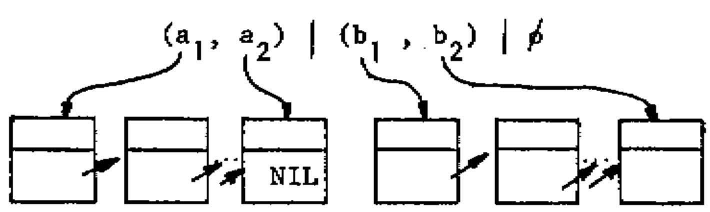


figure 3

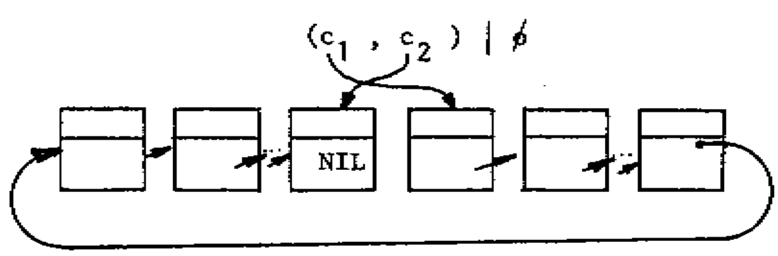


figure 4

Thus, concatenation has consisted of putting the address a_1 in the link of the word pair pointed to by b_2, of replacing the address b_2 by a_2, and of popping the chain accumulator. The use of the chain accumulator is ubiquitous in the list processing operations discussed here. The symbolism $|\phi \rightarrow A\ |\phi$ means that A was stacked on top of the chain accumulator. The symbol ϕ represents whatever was in the chain accumulator previously.

CONSTRUCTIVE OPERATIONS

When the declaration SYMBOL S ; is processed the following code is compiled:

```
CLA    postfix integer for S
TRM    CREATE CHAIN
STL    STORLOC
```

The routine to create a chain for S takes cells from available space and constructs a chain of the form /[CONT:][NAME:S]. As the value of the attribute NAME the relative address of the print name of S is inserted. This relative address is obtained by a transformation on the postfix integer found in the accumulator upon entrance to the routine. The output of the routine is the address of the head of the chain created. The code then stores this address in the location in memory reserved by the compiler for the symbol S. Thus, the value of a symbol variable is the address of the head of its chain.

To construct a list, such as the one in the following example, the compiler produces code as given. For the assignment S $\leftarrow$[A,B,C,D] the code is:

code		effect on chain accumulator
TRM	STACK S	$\mid \emptyset \rightarrow S \mid \emptyset \rightarrow$
TRM	STACK A	$A \mid S \mid \emptyset \rightarrow$
TRM	STACK B	$B \mid A \mid S \mid \emptyset \rightarrow$
TRM	CONCATENATE	$A \frown B \mid S \mid \emptyset \rightarrow$
TRM	STACK C	$C \mid A \frown B \mid S \mid \emptyset \rightarrow$
TRM	CONCATENATE	$A \frown B \frown C \mid S \mid \emptyset \rightarrow$
TRM	STACK D	$D \mid A \frown B \frown C \mid S \mid \emptyset \rightarrow$
TRM	CONCATENATE	$A \frown B \frown C \frown D \mid S \mid \emptyset \rightarrow$
TRM	STORE	$\mid \emptyset$

The last command stores the chain on the top of the chain accumulator into the contents of the item second from the top in the chain accumulator. After the operation S has a value which is the chain /[CONT:A,B,C,D][NAME:S].

To construct and assign the description list S ←/[COLOR:RED][TYPES:MU,RHO]; the following code is produced.

```
TRM  STACK    S
TRM  STACK    COLOR
TRM  MAKE TOP OF CHAINACC AN ATTRIBUTE
TRM  STACK    PURPLE
TRM  CONCATENATE
TRM  STACK    TYPES
TRM  MAKE IT ATTRIBUTE
TRM  CONCATENATE
TRM  STACK    MU
TRM  CONCATENATE
TRM  STACK    RHO
TRM  CONCATENATE
TRM  DESCRIPTION LIST STORE
```

The result of the description list store operation is to change S from /[CONT:A,B,C,D][NAME:S] into /[CONT:A,B,C,D][COLOR:RED][TYPES:MU,RHO][NAME:S].

A final type of constructive operation to be considered is the construction of list structures. Suppose we have the statement

S <-[3,8, TRUE, FxG, J, [A,B,C], <S>],

where F and G are formulas and where J is an integer. Then the code produced will be:

```
TRM   STACK    S
CLA            3.8
TRM   Make ACC into a REAL data term.  Leave address in ACC
STACK    <ACO
CLA            TRUE
TRM   Make ACC into a Boolean data term.  Leave address in ACC
STACK    <ACO
CONCATENATE
Code Piece to construct formula FxG and to leave address of head
  of resulting formula in accumulator
STACK    <ACO
CONCATENATE
CLA     J
TRM   Make ACC into integer data term.  Leave address in ACC
STACK    <ACO
CONCATENATE
STACK   A
STACK   B
CONCATENATE
STACK   C
CONCATENATE
TRM   Make top chain in chain accumulator into a local chain and
             leave address of local chain stacked on top of chain
             accumulator.
CONCATENATE
STACK   S
TAKE CONTENTS
CONCATENATE

STORE
```

It is worthwhile to note that in the absence of the chain accumulator $N\chi(N+1)/2$ search operations are required to build up a chain of length N (assuming as the alternate scheme that we have the address of the head in the accumulator, that we search to the end, and that having found it we append a new element). With the chain accumulator no search operations are needed to find the end of the chain since we have it already stored. The chain accumulator also proves useful when given a chain, we wish to focus some search operation on a subchain whose boundaries we wish to have precisely delimited.

SELECTION EXPRESSIONS

When writing code for selection expressions one must first stack on top of the chain accumulator the chain on which the selection is to be performed, then one must perform the selection leaving the selected sub-chain on top of the chain accumulator. Now it happens that the order in which these two operations must be performed is the reverse of the order in which they are specified in the source language. For example, if one were parsing the expression N TH OF S one would first recognize the selector N TH OF and,second,one would recognize S; yet S must appear on the chain accumulator stack before selection can be performed on it. To implement this flads are used so that the control flow in the code produced can be the reverse of the order of recognition. Thus, for N TH OF S the following code is produced:

```
    TRA   θ
ρ:  CLA   N
    TRM   Selection Routine to get Nth of chain in top of chain acc.
    TRA   χ
θ:  STACK S
    TAKE CONTENTS
    TRA   ρ
χ:  . . .
```

The code corresponding to LAST OF S uses a zero in place of N in the above code.

Consider now the example 3 RD FORMULA OF S. Here we have to search for successive elements of the type FORMULA imbedded in a chain of elements which may include elements other than formulas. The code produced for this is quite similar to the code for N TH OF S. It is as follows:

```
        TRA   θ
    ρ:  CLA   3
        STI   X1
        CLA   Type FORMULA ( ←a bit pattern )
        STI   X2
        TRM   Selection routine for Nth or LAST <type>.
                   leaves integer for position in accumulator as output.
        TRM   Convert integer for position into subchain selection.
        TRA   χ
    θ:  STACK     S
        TAKE CONTENTS
        TRA       ρ
    χ:  . . .
```

The expressions LAST F OF S, 1 ST (|VOWEL|) OF S, and N TH (F + G×3) OF S produce code identical to the code above, except the class name or expression is stored in X2 and a mark transfer to a different selection routine is made.

Another kind of selection expression is exemplified by the following list:

```
FIRST 4 OF S
LAST  3 OF S
ALL BEFORE 3RD SYMBOL OF S
ALL AFTER LAST FORMULA OF S
```

The first and third of these expressions produces a call on the selection routine to select all elements before but not including the Nth element of the chain stacked on top of the chain accumulator. The second and fourth of these expressions produce calls on a selection routine to select all elements after the Nth element of the chain stacked on top of the chain accumulator. Thus, the code for the expression FIRST 4 OF S is as follows:

```
    TRA    θ
ρ:  CLA    4
    ADD    1
    TRM    Select all before <ACC>
    TRA    χ
θ:  STACK      S
    TAKE CONTENTS
    TRA        ρ
χ:  . . .
```

In the case of ALL BEFORE 3RD SYMBOL OF S the code starting at ρ above is replaced with code to compute the location of the third symbol of S and to leave the position as an integer in the accumulator. This consists of using the same type selection routine as was shown in the code sample on page 70 at the top. [This is the reason that an integer was left in the accumulator in the code sample on the top of page 70 even though it may have seemed inefficient at the time. The type selection routine is thus seen to be shared by a number of types of code pieces with different structures and different functions. It is most convenient to have the output of this routine left as the integer giving the position of the object found.]

In the case of the expression LAST 3 OF S the code starting at ρ in the code sample on this page, above, would be replaced with a

```
    TRM    Count length of list on top of chain accumulator.
    SUB    3
    TRM    SELECT ALL AFTER <ACC>
    TRA    χ
```

Likewise in the case of the expression ALL AFTER LAST FORMULA OF S one replaces the code at p with

```
Codepiece to compute position of last formula in chain
on top of chain accumulator.  Position found left as an
integer in normal accumulator.
STI     temp
TRM     COUNT LENGTH OF LIST in chain acc.
SUB     <temp>
TRM     SELECT ALL AFTER <ACC>
```

A more complicated example is the following:

BETWEEN FIRST SYMBOL AND 3RD BEFORE LAST X OF S.

The stratagem for computing subchains between two expressions is to calculate the integer positions in the chain between which the subchain will extend. Then find the greater of the two, take the subchain consisting of all elements before that integer position, then in this subchain take all elements after the integer position which is the lesser of the two. This clearly gives the subchain between the two. The result is that we construct code to compute both integer positions, and we deliver both integers to the BETWEEN SELECTOR routine which does an arithmetic comparison of the two positions and calls the ALL BEFORE and ALL AFTER routines in succession to accomplish its objective.

A final type of selection routine we will consider is the type exemplified by expressions such as ALL SYMBOL OF S and ALL SUBLIST OF S. These expressions can be used in two separate contexts:

First Possibility: L ← [ALL SYMBOL OF <<S>>];

Second Possibility: DELETE ALL SUMBOL OF <<S>>;

In the first possibility the selector routine should leave a concatenated chain consisting of all SYMBOLS found in the chain <<S>>. In the second case the selector routine should leave position markers allowing the deletion routine to perform deletions at each position marker. The situation is resolved by having the ALL SELECTOR ROUTINE leave position markers stacked in the chain accumulator and a check is made in all constructive operations (such as concatenating lists or description lists) to see that any position markers left by the ALL SELECTOR ROUTINE have their referents concatenated into a unit before partaking in a constructive operation. The deletion routine can then perform deletions at each position marker.

EDITING STATEMENTS

Consider the editing statement INSERT [A,B,C] AFTER LAST SYMBOL, BEFORE FIRST (|VOWEL|) OF S. The code produced for this is as follows:

```
       STACK    A
       STACK    B
       CONCATENATE
       STACK    C
       CONCATENATE
       TRA      θ
    ρ: Compute location of last symbol. Find this position in
       the chain and stack an insertion locator pointing to it.
       STACK insertion locator 2 down in chain accumulator
       Compute location of first ( VOWEL ) minus one. Find
       this position in the chain and stack an insertion locator
       pointing to it.
       STACK the insertion locator 2 down in chain accumulator
       TRM      INSERTION ROUTINE
       TRA      χ
    θ: STACK    S
       TAKE CONTENTS
       TRA      ρ
    χ: . . . .
```

Let us now trace the effect of executing this code on the contents of the chain accumulator. We begin in the initial state | ø. Upon entering the code we build up A⌢B⌢C stacked on top of the chain accumulator getting A⌢B⌢C|ø. Then we transfer to θ where we stack S, S|A⌢B⌢C|ø and take its contents <S> | A⌢B⌢C | ϕ. At this point we transfer back to ρ to start computing the insertion locators. We first compute the position of the last symbol in the chain using the type selection routines explained earlier, then we stack a pointer to the element in the chain <S> which is the last symbol. This converts the chain accumulator to look like ρ | <S> | A⌢B⌢C | ϕ. Since we will always need <S> on top of the stack in order to use it in the process of computing insertion locators we stack the insertion locator just computed two down getting <S> | A⌢B⌢C | ρ | ϕ. Then we compute the second insertion locator corresponding to the position of the first (|VOWEL|) minus one, and we stack it on the chain accumulator getting ρ | <S> A⌢B⌢C | ρ | ϕ. This top insertion locator is now stacked two down producing <S> | A⌢B⌢C | ρ | ρ | ϕ. By now the reader sees that we can continue in this fashion to process as many insertion locators as we wish from an insertion locator list of any length. Finally, we come to the INSERTION ROUTINE. This routine pops <S> from the chain accumulator and inserts copies of A B C at every insertion locator looping until all insertion locators in the chain accumulator are exhausted. The state of the chain accumulator after the statement is | ϕ.

The code produced for the DELETION ROUTINE follows a similar strategy. The code stacks selectors pointing to the subchains that are to be deleted. Then a transfer is made to the deletion routine which zeroes out the interiors of the subchains referred to. A final pass removes from the chain all zero elements. Two passes are needed, since it is legal to DELETE two subchains,one of which is overlapping part of the other. If we remove the subchains from the chain as we go along we are in danger of having subsequent subchain deletion operations destroy the integrity of the chain by linking the first part of the chain to available space and by linking the available space to the second part of the chain.

Deletion of interior of this subchain indicated by dotted lines - - -

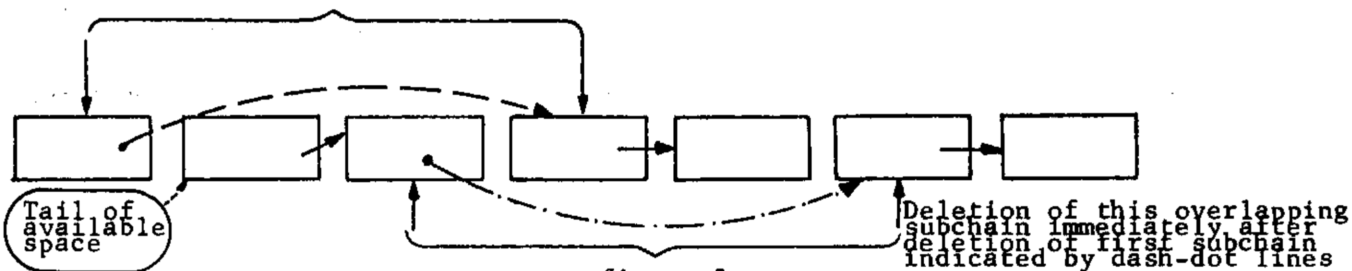


figure 5

Alteration statements such as ALTER (1ST FORMULA, 3RD BEFORE LAST, LAST SYMBOL) OF S TO [A,B,C] again produce code similar in strategy to that produced by the insertion and deletion statements. The selectors are computed and the subchains they point to are stacked. The interiors of these subchains are zeroed out and the insertions are performed by inserting copies of the chain to be inserted after the last zero of the subchains zeroed out. Finally, the zero elements are erased. An attempt to set up alteration with less passes leads to destruction of the integrity of the chain in some cases of overlap. Thus, the multiple passes are necessary. The description list editing statements THE A OF B IS NOT C and THE A OF B IS ALSO C are special cases of deletion and insertion. The first computes the subchain consisting of the value list THE A OF B and applies the operation DELETE C to it. The second checks to see if C is among the value list THE A OF B and does an INSERT C AFTER LAST OF to the value list should it be the case that C was not on it beforehand.

PUSH DOWN AND POP UP STATEMENTS

A push down statement merely inserts a bar attribute | between the contents attribute and the first element after the contents attribute. For example, if we have executed S ← [A,B,C] then the chain in S looks like /[CONT:A,B,C][NAME:S]. Then executing ↓S causes the following code to be compiled:

```
STACK S
TRM  PUSH DOWN ROUTINE
```

Where the latter routine changes the chain in S to look like /[CONT:] [|:A,B,C] [NAME:S]; The pop up operation is the inverse of this deleting the contents and removing the first bar attribute | found after the contents. The code for pop up is

```
STACK  S
TRM  POP UP ROUTINE
```

FOR STATEMENTS

Suppose we execute L «- [A,B,C] and then encounter the statement FOR S <-ELEMENTS OF L DO.... This causes the following code to be compiled:

```
    STACK  S                        \J> - > S J / -»
    STACK  L                        L| S I 4> -»
    TAKE CONTENTS                   <L> | S { ^ -»
    COPY TOP OF CHAINACC            copy (<L» | S | j -»
CT: TRM FOR LIST GENERATOR
    TRA  9
    TRM  p
    TRA  a

          *_______closed subroutine for body of for-statement

01 • • •
```

When the for list generator is called it detaches the first element of the copy of L found on top of the chain accumulator and inserts this first element in S. It then exits green causing a mark transfer to the closed subroutine for the body of the for statement and upon return control passes back to the for list generator for another iteration. On successive iterations it detaches the successive elements of the copy of L and places them in the contents of the control variable. Finally, the copy of L becomes exhausted, and the for list generator exits red, causing it to transfer around the code for the for statement body.

In the case of parallel for statements, such as

PARALLEL FOR (I,J,K) $\leftarrow$ ELEMENTS OF (<S>,<T>,<U>) DO..

the generator stacks a list of the control variables I,J, and K, and a list of sublists [<S>, <T>, <U>], each sublist being a copy of the original. The generation cycle detaches each control variable and its corresponding sublist, stacks them, calls the simple for list generator explained above, and returns them when finished. The generation stops on the first cycle before all sublists are exhausted. The control structure is identical to that explained above.

IDENTITY ROUTINES

There is a recursive identity routine which accepts its two parameters as chains stacked on the chain accumulator and which outputs a true or false in the normal accumulator.

PASSING ACTUAL PARAMETERS

The thunks for actual parameters which are symbolic expressions stack their arguments on the chain accumulator when called.

APPENDIX I

TABLE 1 - PRODUCTIONS

Label	Stack	Input		Replacement		Action	Next
ENTER THE ALGOL TRANSLATOR							
S0	|→	BEGI	| →	|→	|		*D1
+1		<SG>	|		|	ERROR 0	Q0
'BEGIN' HAS BEEN SCANNED.							
D1	|→	<DC>	| → BEGI |→	<DC>	|	EXEC 1	
						SUBR DEC	S1
+1	|→	<SG>	| → BEG* |→	<SG>	|		S1
+2		<SG>	|		|	ERROR 25	Q25
ENTER STATEMENT SCAN WITH TERMINATOR IN STACK							
S1	|→	<SG>	|		|		S1A
+1		<SG>	|		|	ERROR 98	Q98
S1A		BEGI	| →		|		*D1
+1		FOR	|		|		*E1
+2		IF	|		|		*E1
+3		GO	|		|		*G1
+4		;	|		|		E30
+5		END	|		|		E30
+6		I	| →	E	|		*S2
+7		PRIN	| →		|		*S3A
+8		<	|		|		*E1
+9		(	| →	E(	|		*E1
+10		<SL>	|		|	EXEC 193	SL2
+11		THE	|		|		*E1
+12		↓	|		|		*PD1
+13		↑	|		|		*PU1
+14		INSE	|		|		*E1
+15		DELE	|		|		*SL0
+16		ALTE	|		|		*SL0
+17		PARA	|		|	SCAN	*PF1
+18		LET	|		|	SCAN	*CL1
+19		<SG>	|		|	ERROR 1	Q1
AN IDENTIFIER HAS BEEN SCANNED AT THE BEGINNING OF A STATEMENT.							
S2	E	:	| →		|	EXEC 91	*S1
+1	E	(	|		|	SUBR COL	S2A
+2	E	[	|		|	EXEC 16	*E1
+3	E	←	| →	E ←←	|	EXEC 9	
						EXEC 66	
						EXEC 211	*E1
S2A	E	END	| →	END	|	EXEC 10	E30
+1	E	ELSE	| →	ELSE	|	EXEC 10	E25
+2	E	;	| →	;	|	EXEC 10	E30
+3	E	<OS>	| →	OSE	|	EXEC 193	
						EXEC 47	*OS1
+4		<SG>	|		|	ERROR 2	Q2
S3A		(	| →	PR(	|		*E1
+1		<SG>	|		|	ERROR 75	Q0

EXPRESSION SCANNER - THE GUTS OF THE TRANSLATOR

```
    EXPRESSION SCANNER PART 1:  OPERAND EXPECTED
E1                              I     | →            E     |                     *E2
 +1      <      E      >        <SG>  | →      E     <SG>  |     EXEC 97         CON
 +2             E      /        [     | →      E     /[    |                     *E1H
 +3                             +     | →                  |                     *E1
 +4                             -     | →            NG*   |                     *E1
 +5                             <UN>  |                    |                     *E1A
 +6                             ↓     |                    |                     *E1
 +7                             ¬     |                    |                     *E1
 +8                             (     | →            E(    |                     *E1
 +9                             IF    |                    |                     E20
+10                             <BI>  | →            E     |     EXEC 8          *E2
+11                             B                          |     SCAN            *E2G
+12                             DERV  | →                  |                     *E1B
+13                             EVAL  |                    |     SCAN            *EV1
+14                             OF    |                    |                     *E1C
+15                     =       =     | →            INST  |                     E2A
+16                     >       >     | →            CONT  |                     E2A
+17                             <TP>  |                    |     SUBR CNG        *E1D
+18                             NIL   | →            E     |     EXEC 182        *E2A
+19                             $     |                    |                     *$1
+20                             .     |                    |                     *E2F
+21                     ←←      /     |                    |                     *E1E
+22                     /       [     |                    |                     *E1H
+23                             <     |                    |                     *E1
+24                             <SL>  |                    |     EXEC 193        SL2
+25                             THE   |                    |                     *E1
+26                             [     | →            L[    |                     *E1
+27                             <EA>  |                    |     SCAN            *E1
+28                             DL    |                    |     SCAN            *E1
+29                             '     |                    |                     *TX1
+30                             <SG>  |                    |     ERROR 3         03

    A UNARY OPERATOR HAS BEEN SCANNED.
E1A                             (     | →            E(    |                     *E1
 +1             (      <SG>     ,     | →            (     |     EXEC 13         *E1
 +2             (      <SG>     )     | →            E     |     EXEC 13         XXX
 +3                             <SG>  |                    |     ERROR 41        041
E1B                             (     | →            D(    |                     *E1
 +1                             <SG>  |                    |     ERROR 76        00
E1C                    OF       (     | →            OF(   |                     *E1
 +1                             <SG>  |                    |     ERROR 76        00
E1D                    TYPE     <SG>  | →      E     <SG>  |     EXEC 83         E2A
 +1                             <SG>  |                    |     ERROR 99        099
E1E                    /        [     | →            /[    |                     *E1H
 +1                             <SG>  |                    |     ERROR 101       0
E1F                    /[       <OP>  | →            /[    |     EXEC 69         *E1G
 +1                    /[       <UN>  | →            /[    |     EXEC 69         *E1G
 +2                             <SG>  |                    |     ERROR 80        00
E1G                    /[       ,     | →            /[    |                     *E1F
```

```
 +1              /[    ]    | →     /[   EL    |                           *EL1
 +2                    <SG> |                  |          ERROR 81         Q0
E1H                    COMM | →          E     |          EXEC 195         *E2A
 +1                    CONT | →          E     |          EXEC 188         *E2A
 +2                    INDE |                  |          EXEC 196         *E1J
 +3                    OPER | →          E     |          EXEC 208
                                                          SCAN
                                                          NSTK 2           E1I
 +4                    <SG> |                  |                           E1
E1I              /[    /[   | →          /[    |          EXEC 92          *E1F
 +1                    /[   |                  |                           *E1F
E1J      /[   /[  INDE :    | → /[  INDE :     |          EXEC 92
                                                          SCAN             *E1K
 +1           /[  INDE :    |                  |          SCAN             *E1K
 +2                    <SG> |                  |          ERROR 115        Q
E1K      INDE :   I    ]    | →          EL    |          EXEC 7
                                                          EXEC 179         *EL1
 +1                    <SG> |                  |          ERROR 115        Q


   EXPRESSION SCANNER PART 2:  OPERATOR EXPECTED
E2            FV   E    (    |                 |          EXEC 7           *EV4
 +1                E    (    |                 |          SUBR CAL
                                                          EXEC 21          E2A
 +2                E    [    |                 |          EXEC 65          *E1
 +3           (    E    ,    | →        (      |          EXEC 12          *E1
 +4           (    E    )    | →        E      |          EXEC 12          XXX
 +5                     .    |                 |                           *E2E
 +6                E    ←    |                 |          EXEC 9
                                                          EXEC 66          E2A
 +7                E    <SG> |                 |          EXEC 7
                                                          EXEC 66          E2A
E2A                     <OP> |                 |  E2B     SUBR COM         *E1
 +1                     <ST> |                 |          SUBR H39         E5
 +2                     ,    |                 |          SUBR COM         E11
 +3                     ]    |                 |          SUBR COM         E3
 +4                     THEN |                 |          SUBR COM         E21
 +5                     ELSE |                 |          SUBR COM         E25
 +6                     ;    |                 |          SUBR COM         E30
 +7                     END  |                 |          SUBR COM         E30
 +8                     )    |                 |          SUBR COM         E6
 +9                     STEP |                 |          SUBR COM         F10
+10                     UNTI |                 |          SUBR COM         F15
+11                     WHIL |                 |          SUBR COM         F20
+12                     DO   |                 |          SUBR COM         F31
+13                     :    |                 |          TEST /[          E2C
                                                          NEXT Q
                                                          TEST L[          E2C
                                                          NEXT Q
                                                          TEST X[          E2C
                                                          NEXT Q           E2D
                                                  E2C     SUBR COM         E4
+14                     B    |                 |          SCAN             *E2G
+15                     <SM> |                 |          SUBR H39         E2H
+16                     ¬    | →               |                           *NR
```

UTILITY ROUTINES FOR THE EXPRESSION SCANNER

```
    RETURN FROM COM AFTER <SM> HAS BEEN SCANNED
E2H                  E    <OS> | →             OSE   |  EXEC 193
                                                        EXEC 47     *OS1
 +1             ALL  E    <PE> | →       SL    <PE>  |  EXEC 96
                                                        EXEC 48     SL1
 +2                  E    IS   |                     |              *IS1
 +3                  E    HAS  |                     |              *E1
 +4             OSE  E    <PE> | →       EP    <PE>  |  EXEC 96
                                                        EXEC 166    EP1
 +5             ALTE E    TO   |                     |              *E1
 +6                  E    IN   |                     |  SCAN        *CL1
 +7        OSE  INTE E    <PE> | →       EP    <PE>  |  EXEC 177    EP1
 +8             [    E    B    |                     |              *E1
 +9                  E    B    |                     |              *CL2
+10             THE  E    OF   |                     |              *E1
+11             INSE E    <SG> |                     |  EXEC 193    ILO
+12                       <SG> |                     |  ERROR 100   0

    ')' HAS BEEN SCANNED.  THE STACK SHOULD CONTAIN THE MATCHING '['.
E3   GO   E     [    E    ]    | →                   |  EXEC 15     *G4
 +1       E     [    E    ]    | →             E     |  EXEC 25
                                                        EXEC 17     *E3B
 +2  A[   E     :    E    ]    | →                   |  EXEC 141
                                                        EXEC 145    *ART
 +3       (     [    E    ]    | →       (     E     |  EXEC 64     *EV2
 +4             L[   E    ]    | →             EL    |  EXEC 194    *EL1
 +5  L[   EL    ,    E    ]    | →             EL    |  EXEC 194
                                                        EXEC 92     *EL1
 +6  CN   E     B    E    ]    | →                   |              *E2
 +7             Ev(  E    ]    | → E(    E     )     |  EXEC 74     E6
 +8             .(   E    ]    | →             E     |  EXEC 73
                                                        EXEC 95     *E2A
 +9                       <SG> |                     |  ERROR 5     Q5
E3B             (    E    ,    | →             (     |  EXEC 24     *E1
 +1             (    E    )    | →             E     |  EXEC 18     XXX
 +2             E(   E    )    | →             E     |  EXEC 64     *E3B
 +3                  E    <ST> |                     |              E5
 +4             PR(  E    ,    | →             PR(   |  EXEC 99     *E1
 +5             PR(  E    )    | →                   |  EXEC 99     *E44
 +6             EVAL E    <SG> | →       E     <SG>  |  EXEC 70     E2A
 +7             EV(  E    ,    | →             EV(   |  EXEC 73     *E1
 +8             EV(  E    )    | → (     E     )     |  EXEC 74     EV2
 +9             FV   E    (    |                     |              *EV4
+10             B    E    B    | →             CLSO  |  EXEC 64
                                                        SUBR COM    EB1
+11             |→   E    <SG> |                     |  ERROR 6     Q6
+12                       <SG> |                     |              E2A

         FUNCTION CALL HAS BEEN SCANNED
XXX                  E    E    | →             E     |  EXEC 20     *RET
 +1                  <UN> E    | →             E     |  EXEC 14     *RET
```

```
 +2                       <SG> |                    |       ERROR 14     00

   ':' HAS BEEN SCANNED
E4             X(   E     :    | → A(   E    :      |                    *E1
 +1       /(   /(   E     :    | → /(   L(   EL     |       STAK ,
                                                            EXEC 175
                                                            EXEC 92      *E1
 +2            /(   E     :    | → /(   L(   EL     |       STAK ,
                                                            EXEC 175     *E1
 +3                 E     :    | →      E    ,      |       EXEC 186     E11
 +4                       <SG> |                    |       ERROR 7      07

   '←' OR ':=' HAS BEEN SCANNED.
E5             |→   E     ←    | → |→   E    ←←     |       EXEC 211     *E1
 +1            FOR  E     ←    | → FOR  E    ←←     |       EXEC 211
                                                            EXEC 212
                                                            EXEC 39      *E1
 +2            PARA E     ←    | →      PARA ←      |       SCAN
                                                            SCAN         *PF2
 +3       FOR  <ST> E     <SG> |                    |       ERROR 8      08
 +4                 E     <ST> |                    |       EXEC 211     *E1
 +5                       <SG> |                    |       ERROR 8      08

   ')' HAS BEEN SCANNED.
E6        FV   E(   E     )    | →      FV   E      |       EXEC 64      *EV4
 +1            E(   E     )    | →           E      |       EXEC 64      *E2A
 +2            PR(  E     )    | →                  |       EXEC 99      *E44
 +3            .(   E     )    | →           E      |       EXEC 73
                                                            EXEC 95      *E2A
 +4            OF(  E     )    | →           E      |       EXEC 84      *E2A
 +5            (    E     )    | →           E      |       EXEC 18      XXX
 +6            EV(  E     )    | → (    E    )      |       EXEC 74      EV2
 +7 D(    E    ,    E     )    | →           E      |       EXEC 98      *E2A
 +8                 E     )    |                    |                    E2H
 +9       DL   (    E     )    | →           E      |       EXEC 189     *E2A
+10                       <SG> |                    |       ERROR 9      09

   ',' HAS BEEN SCANNED
E11            (    E     ,    | →           (      |       EXEC 24      *E1
 +1            PR(  E     ,    | →           PR(    |       EXEC 99      *E1
 +2            D(   E     ,    |                    |                    *E1
 +3            <BK> E     ,    | →           <BK>   |       EXEC 73      *E1
 +4 A(    E    :    E     ,    | →           X(     |       EXEC 141     *E1
 +5            X(   E     ,    |                    |       ERROR 42     042
 +6 GO    E    [    E     ,    |                    |       ERROR 39     039
 +7            [    E     ,    | →           [      |       EXEC 25      *E1
 +8       UNTI FOR  E     ,    | →           FOR    |       EXEC 26      E12
 +9       WHIL FOR  E     ,    | →           FOR    |       EXEC 27      E12
+10                 FOR   ,    | →           FOR    |       EXEC 28      E12
+11            L(   E     ,    | → L(   EL   ,      |       EXEC 194     *E1
+12 L(    EL   ,    E     ,    | → L(   EL   ,      |       EXEC 194
                                                            EXEC 92      *E1
+13                 E     ,    |                    |                    E2H
+14                       <SG> |                    | EA13  ERROR 10     010
E12                       <SG> | → <SG> E    ←←     |       EXEC 29      *E1
```

```
•17                       <SG>  1                   |      ERROR  4     04
NR                        <     1 •*           NL   1                   E2B
 •1                       >     1 •»           NG   1                   E2B
 •2                       <S6>  1                   |      ERROR  4     04
E2D                       t     1 •*           LI   |      EXEC  76
                                                           SUBR  COM    •EI
 •1                       <SQ>  1                   |      ERROR  99    099
E2E                 •           1 «*           •«•  1      SUBR  COM    •EI
 •1            ∃    •     t     1              .(   1      EXEC  94     •EI
 +2                       <SG>  1                   |      ERROR  77    00
E2F            t    I     <SG>  1         E    <S6> |      EXEC  7
                                                           EXEC  47     E2A
 +1            •    IF    <SG>  1 •*      .IF  <SG> |                   EI
 +2                       <SG>  1                          ERROR  77    00
E2G            B    I     8     1 •*           CLSO 1      EXEC  64
                                                           SUBR  COM
                                                      EB1  EXtC  75     •EI
 +1            8    I     t     » •* 0    t    I    1      EXEC  65     •EI
 +2                       <SG>  1                   1      ERROR  78    00
```

```
        TYPES
INT               OSE  INTE <PE> | → OSE  TYPE <PE> |       EXEC 147     RT1
 +1               OSE  INTE <SG> |                  |                    E1
RT1                    TYPE <SG> |                  |       EXEC 83      RT2
RT2               OSE  TYPE <SG> | →      EP   <SG> |       EXEC 181     EP1
 +1               ALL  TYPE <SG> | →      SL   <SG> |       EXEC 200     SL1
 +2                         <SG> |                  |       ERROR 116    0

        PUSH AND POP
PD1                    ↓    ↓    |                  |                    *PD1
 +1                    ↓    <SG> |                  |                    E1
 +2                         <SG> |                  |       ERROR 113    0
PU1                    ↑    ↑    |                  |                    *PU1
 +1                    ↑    <SG> |                  |                    E1
 +2                         <SG> |                  |       ERROR 114    0

        TREE EXPRESSIONS AND DESCRIPTION LISTS
EL1               /(   EL   (    | →      /(   /(   |                    *E1H
 +1 E      ←←     /(   EL   <SG> | →           <SG> |       EXEC 110     E2A
 +2        E      /(   EL   <SG> | →      E    <SG> |       EXEC 103     E2A
 +3                    EL   /    |                  |                    *EL2
 +4        EL     /(   EL   <SG> | →      E    <SG> |       EXEC 173     E2A
 +5        E      ←    EL   <SG> | →      EL   <SG> |       EXEC 176
                                                            EXEC 104     EL1
 +6        E      ←←   EL   <SG> | →           <SG> |       EXEC 176     E2A
 +7               INSE EL   <SG> | → INSE E    <SG> |       EXEC 193     IL0
 +8        E      INST EL   <SG> | →      E    <SG> |       EXEC 183     E2A
 +9               PARA EL   <SG> | →      PARA <SG> |       SCAN
                                                            SCAN         *PF2
 +10 |→    PARA ←← EL   DO   | →      DO   |→   |       EXEC 217     FA33
 +11                   EL   <SG> | →      E    <SG> |       EXEC 173     E2A
 +12                        <SG> |                  |       ERROR 102    0
EL2                    /    (    | →      /(   /(   |                    *E1H
 +1                         <SG> |                  |       ERROR 101    0

        TEXT
TX1               '    EL   '    | →           E    |       EXEC 218     *E2A
 +1                         '    |                  |                    *TX2
 +2                         I    |                  |                    *TX2
 +3                         <SG> | →           E    |       EXEC 202     *TX3
TX2                    .    I    |                  |                    *TX2
 +1               '    I    <SG> | →      E    <SG> |       EXEC 7
                                                            EXEC 47      TX3
 +2                    I    <SG> | →      E    <SG> |       EXEC 7
                                                            EXEC 66      TX4
 +3                         <SG> |                  |       ERROR 117    0
TX4                    E    <SG> |                  |       EXEC 194     TX3
TX3               EL   E    <SG> | →      EL   <SG> |       EXEC 92      TX1
 +1                    E    <SG> | →      EL   <SG> |                    TX1
```

```
         MISCELLANEOUS
IS1                         IS   NOT   |                    |                *E1
 +1                         IS   ALSO  |                    |                *E1
 +2                         IS   <SG>  |                    |                E1
 +3                              <SG>  |                    |   ERROR 103    0
$1                          $    <,]>  |  →      E     <,]> |   EXEC 185     E2A
 +1                              <SG>  |                    |                E1
PF1               PARA FOR       [     |  →      PARA L[    |                *E1
 +1               PARA FOR       <SG>  |  →      PARA <SG>  |                E1
 +2                              <SG>  |                    |   ERROR 116    0
PF2          ←    ELEM OF        [     |  →      ←←   L[    |                *E1
 +1          ←    ELEM OF        <SG>  |  →      ←←   <SG>  |                E1
 +2                              <SG>  |                    |   ERROR 116    0
SL3               ALTE E         <SG>  |                    |                RET
 +1               DELE E         <SG>  |  →           <SG>  |   EXEC 59      RET
 +2                    E         <SG>  |                    |   EXEC 207
                                                                EXEC 178     COM
VR1               |→   E         IS    |                    |   RETURN
 +1               DELE E         <SG>  |  →           <SG>  |   EXEC 219     RET
 +2                              <SG>  |                    |   EXEC 178     COM
CON               DELE E         <SG>  |  →           <SG>  |   EXEC 219     E2A
 +1                    E         <SG>  |  →                 |   EXEC 178     E2A
```

MORE UTILITY ROUTINES FOR THE EXPRESSION SCANNER

```
     SELECTORS
OS1                 OSE  INTE |                 |                *INT
 +1                 OSE  <TP> |                 |    SUBR CNG    *RT1
 +2                 OSE  (    | →    OSE  E(    |                *E1
 +3                 OSE  <BA> |                 |                *P00
 +4                 OSE  <PE> | →    EP   <PE>  |    EXEC 206    EP1
 +5                 OSE  <SG> |                 |    EXEC 209    E1
 +6                      <SG> |                 |    ERROR 104   0
EP1       OSE  BEFO EP   <SG> | →    PO   <SG>  |    EXEC 53     P1
 +1       OSE  AFTE EP   <SG> | →    PO   <SG>  |    EXEC 54     P1
 +2                 EP   <SG> | →    PO   <SG>  |                P1
 +3                      <SG> |                 |    ERROR 105   0
P1             BETW PO   AND  |                 |                *P00
 +1  BETW PO   AND  PO   <SG> | →    SL   <SG>  |    EXEC 55     SL1
 +2       ALL  BEFO PO   <SG> | →    SL   <SG>  |    EXEC 56     SL1
 +3       ALL  AFTE PO   <SG> | →    SL   <SG>  |    EXEC 57     SL1
 +4            AFTE PO   OF   | →         IL    |    EXEC 205    *IL1
 +5            BEFO PO   OF   | →         IL    |    EXEC 204    *IL1
 +6                 PO   <SG> | →    SL   <SG>  |    EXEC 191    SL1
 +7                      <SG> |                 |    ERROR 106   0
SL1                 SL   OF   |                 |    EXEC 192    *E1
 +1       SL   ,    SL   <SG> | →    SL   <SG>  |                SL1
 +2                 SL   ,    |                 |                *SL0
 +3            E(   SL   )    | →         SL    |                *SL1
 +4                      <SG> |                 |    ERROR 107   0
IL1                 IL   ,    |                 |                *IL0
 +1       IL   ,    IL   <SG> | →    IL   <SG>  |                IL1
 +2            (    IL   )    | →         IL    |                *IL1
 +3                 IL   <SG> |                 |    EXEC 58
                                                     EXEC 192    E1
 +4                      <SG> |                 |    ERROR 108   0
IL0                      <BA> |                 |                *P00
 +1                      (    |                 |                *IL0
 +2                      <SG> |                 |    ERROR 109   0
AL1                 ALL  <BA> |                 |                *P00
 +1                 ALL  <TP> |                 |    SUBR CNG    *RT1
 +2                 ALL  <SG> |                 |    EXEC 209    E1
 +3                      <SG> |                 |    ERROR 110   0
SL0                      <SL> |                 |    EXEC 193    SL2
 +1                      <SG> |                 |    ERROR 111   0
SL2                      BETW |                 |                *P00
 +1                      (    | →         E(    |                *SL2
 +2                      ALL  |                 |                *AL1
P00                      I    | →         E     |                *E2
 +1                      (    | →         E(    |                *E1
 +2                      FIRS | →         OSE   |    EXEC 67     *OS1
 +3                      LAST | →         OSE   |    EXEC 46     *OS1
 +4                      <SG> |                 |    ERROR 112   0
```

```
    'IF' SCANNED
E20                  THEN IF    |                |      NOTE 3       *E1
 +1             THEN |→   IF    |                |      ERROR 38     038
 +2                  <OP> IF    |                |      NOTE 4       *E1
 +3                       <SG>  |                |                   *E1    1

    'THEN' SCANNED
E21        |→   IF   E    THEN  | →     THEN |→  |      EXEC 30      *S1    1
 +1        GO   IF   E    THEN  | →     THEN GO  |      EXEC 30      *G1    1
 +2             IF   E    THEN  | →          THEN|      EXEC 30      *E1    1
 +3             .IF  E    THEN  | →          .THN|      EXEC 81      *E1
 +4                       <SG>  |                |      ERROR 11     011    1

    'ELSE' SCANNED.
E25             THEN E    ELSE  | →          ELSE|      EXEC 38      *E1
 +1             .THN E    ELSE  | →          .ELS|      EXEC 88      *E1

    'ELSE' SCANNED AFTER 'END' OR AFTER 'GO TO IF ... '
E26             THEN |→   ELSE  | →     ELSE |→  |      EXEC 31      *S1
 +1             THEN E    ELSE  | →          ELSE|      EXEC 31      *E1
 +2        DO   |→   E    ELSE  | →     |→   ELSE|      NOTE 7              1
                                                        EXEC 32      E26    1
 +3                       <SG>  |                |      ERROR 12     012    1

    'END' OR ';' HAS BEEN SCANNED.
E30             THEN |→   <SG>  | →     |→   <SG>|      EXEC 33      E30    1
 +1             ELSE |→   <SG>  | →     |→   <SG>|      EXEC 34      E30    1
 +2             DO   |→   <SG>  | →     |→   <SG>|      EXEC 32      E30    1
 +3        RECU PROC |→   ;     | →              |      EXEC 19      *CNT
 +4             PROC |→   ;     | →              |      EXEC 35      *CNT
 +5                  |→   ;     | →          |→  |                   *S1    1
 +6        |→   BEGI |→   END   | →     |→   |→  |      EXEC 36             1
                                                        HALT                1
 +7             BEGI |→   END   | →          |→  |      EXEC 37      *E43   1
 +8             BEG* |→   END   | →          |→  |                   *E43   1
 +9             PROC |→   <SG>  |                |      ERROR 28     00
+10                       <SG>  |                |      ERROR 13     013    1

    AN 'END' HAS BEEN FOUND AND THE MATCHING 'BEGIN' REMOVED FROM THE STACK.
E43             PROC |→   ;     | →              |      EXEC 35      *CNT

    E44 IS ENTERED AFTER PROCESSING A PROCEDURE STATEMENT.
E44                       END   |                |                   E30    1
 +1                       ELSE  |                |                   E26    1
 +2                       ;     |                |                   E30    1
 +3                       <SG>  |                |      ERROR 14     014    1
```

```
               'FOR' STATEMENT


   'STEP' HAS BEEN SCANNED
F10                FOR  STEP | ->      STEP FOR  |        EXEC 40     F10A
 +1                     <SG> |                   |        ERROR 17    Q17
F10A                    <SG> | -> <SG> E    <-<- |        EXEC 60     *E1

   'UNTIL' HAS BEEN SCANNED.
F15           STEP FOR  UNTI | ->      UNTI FOR  |        EXEC 41     F15A
 +1                     <SG> |                   |        ERROR 18    Q18
F15A                    <SG> | -> <SG> E    -    |        EXEC 61     *E1

   'WHILE' HAS BEEN SCANNED.
F20           STEP FOR  WHIL | ->      WHIL FOR  |        EXEC 42     *E1
 +1                FOR  WHIL | ->      WHIL FOR  |                    *E1
 +2                     <SG> |                   |        ERROR 19    Q19

   'DO' HAS BEEN SCANNED.
F31 |->  UNTI FOR  E    DO   | ->      DO   |->  |        EXEC 26     FA33
 +1 |->  WHIL FOR  E    DO   | ->      DO   |->  |        EXEC 27     FA33
 +2             |->  FOR  DO   | ->      DO   |->  |        EXEC 28     FA33
 +3 |->  PARA <-<- E    DO   | ->      DO   |->  |        EXEC 217
                                                     FA33 EXEC 43     *S1
 +4                     <SG> |                   |        ERROR 20    Q20
```

```
              'GO TO' STATEMENT

   'GO' HAS BEEN SCANNED IN S1
G1                     I    | →          E     |                   *G
 +1               GO   (    | →      G(  GO    |                   *G
 +2          THEN GO   IF   |                  |     NOTE 5        *E
 +3                    IF   |                  |                   *E
 +4                    <SG> |                  |     ERROR 21      Q2

   'GO TO <IDENTIFIER>' HAS BEEN SCANNED
G2                E    (    |                  |                   *E
 +1          GO   E    <SG> | →          <SG>  |     EXEC 44       G4
 +2                    <SG> |                  |     ERROR 44      Q4

   'GO TO <DESIGNATIONAL EXPRESSION>' HAS BEEN PROCESSED AND NEXT SCAN
G4                THEN ELSE | →      ELSE GO   |     EXEC 31       *G
 +1          THEN |→   ELSE | →      ELSE |→   |     EXEC 31       *S
G5                ELSE <SG> | →          <SG>  |     EXEC 34       G5
 +1               SWIT <SG> | →          <SG>  |                   D2
 +2               THEN ;    |                  |     ERROR 22      Q2
 +3                    ;    |                  |                   E3
 +4                    END  |                  |                   E3
 +5                    ELSE |                  |                   E2
 +6               G(   )    | →                |                   *G
 +7                    <SG> |                  |     ERROR 24      Q2
```

DECLARATIONS

```
DEC                          OWN  |                           |        EXEC 156   *TP
TP                           <TP> |                           |        SUBR CHG   *SEC
SEC                          ARRA |                           |                   AR
 +1                     TYPE RECU |  →       RECU TYPE        |        EXEC 158   *SEK
SEK                          PROC |                           |        EXEC 159   *PRI
 +1                     |→   SWIT |                           |                   *SWI
 +2                     LABE I    |  →       TYPE I           |        EXEC 154   TID
 +3                          I    |                           |        EXEC 174
                                                                TID    SUBR ID    CUP
 +4                          <SG>.|                           |        ERROR 174  QDC
CUP               OWN   TYPE ;    |  →                        |        EXEC 139   *CNT
 +1                     TYPE ;    |  →                        |                   *CNT
AR                      TYPE ARRA |  →            ARRA        |        EXEC 142   IDA
 +1                          ARRA |                           |        EXEC 143
                                                                IDA    SUBR SID   ARD
ARD                          [    |  →            X[          |        EXEC 140   *E1
 +1                          <SG> |                           |        ERROR 144  QDC
ART                     ARRA ;    |  →                        |                   *CNT
 +1                     ARRA ,    |  →            ARRA        |        EXEC 144   IDA
 +2                          <SG> |                           |        ERROR 145  QDC
PRI                     PROC I    |  →            P-ID        |        EXEC 160   FND
FND                     TYPE P-ID |  →            P-ID        |        EXEC 161   PSB
 +1                          <SG> |                           | PSB    EXEC 162   *FPL
FPL                          (    |                           |        EXEC 157
                                                                       SUBR SID   PCC
 +1                     P-ID ;    |  →       PROC |→          |        EXEC 163   *S1
 +2                          <SG> |                           |        ERROR 163  QSP
PCC                          )    |  →                        |                   *CCA
 +1                          <SG> |                           |        ERROR 194  QSP
CCA                     (    ;    |  →                        |                   *VAL
CCC                          :    |  →                        |                   *CCB
 +1                          <SG> |  →                        |                   *CCC
CCB                          (    |  →                        |        SUBR SID   PCC
 +1                          <SG> |  →                        |                   *CCC
VAL                          VALU |                           |        EXEC 172
                                                                       SUBR SID   VLU
SP                           <SP> |                           |        SUBR CHG   SPA
 +1                     P-ID <SG> |  →  PROC |→       <SG>    |        EXEC 164   S1
 +2                          <SG> |                           |        ERROR 164  QSP
VLU                     VALU ;    |  →                        |                   *SP
 +1                          <SG> |                           |        ERROR 195  QSP
SPA                          TYPE |                           |        EXEC 167   *SP2
SP2                          I    |                           | ISP    SUBR ID    SPT
 +1                          ARRA |                           |        EXEC 168   *ISP
 +2                          PROC |                           |        EXEC 169   *ISP
 +3                          LABE |                           |        EXEC 170   *ISP
 +4                          SWIT |                           |        EXEC 171   *ISP
 +5                          <SG> |                           |        ERROR 171  QSP
SPT                     TYPE ;    |  →                        |                   *SP
 +1                TYPE <SG> ;    |  →                        |                   *SP
```

```
 +2              <SP> ;   | →               |           ERROR 196   *SP
 +3                   <SG> |                |           ERROR 196   OSP
CNT                   <DC> |                |                       DEC
 +1                   <SG> |                |           EXEC 165    RET

        TYPE  CONVERSION
CNG                   SUBL | →        TYPE  |           EXEC 180    RET
 +1                   ATOM | →        TYPE  |           EXEC 82     RET
 +2                   TEXT | →        TYPE  |           EXEC 201    RET
CHG                   REAL | →        TYPE  |           EXEC 146    RET
 +1                   INTE | →        TYPE  |           EXEC 147    RET
 +2                   BOOL | →        TYPE  |           EXEC 148    RET
 +3                   LOGI | →        TYPE  |           EXEC 149    RET
 +4                   FORM | →        TYPE  |           EXEC 150    RET
 +5                   SYMB | →        TYPE  |           EXEC 151    RET
 +6                   HALF | →        TYPE  |           EXEC 152    RET
 +7                   STRI | →        TYPE  |           EXEC 153    RET
 +8                   <SG> |                | RET       RETURN      IMP

        IDENTIFIER  LIST
ID                    I    | →              |           EXEC 190    *AI
 +1                   <SG> |                |           ERROR 190   AID
AID                   ,    | →              | SID                   *ID
 +1                   <SG> |                |           RETURN      IMP
                                              SWI       SCAN        SW1
SW1          SWIT I   <ST> | →    SWIT GO   |           EXEC 50     *G1
 +1                   <SG> |                |           ERROR 250   G0
D25                   ,    | →    SWIT GO   |           EXEC 51     *G1
 +1                   ;    | →              |           EXEC 52     *CN
 +2                   <SG> |                |           ERROR 251   G0
```

ROUTINE FOR COMPILATION

COM					L:							H38
+ 1					<UN>							W36
+ 2					+							M36
+ 3					t							H34
+ 4					NG*							H32
+ 5					»							H30
+ 6					/							H30
+ 7					+							H28
+ 8												H28
+ 9					<RE>							H26
+ 10												W24
+ 11					's							H22
+ 12					v							H20
+ 13					CLSO							HA1
+ 14					<PN>							W19
+ 15					<0T>							H16
H16		E		E	<SG>			<SG>	|	EXEC	112	RET
+ 1		E	«'	E	<SG>		Ξ	<SG>	|	EXEC	112	
										EXEC	113	COM
+ 2	INSE	E	IL	E	<SG>			<SG>	|	EXEC	63	RET
+ 3	ALTE	E	TO	E	<SG>			<SG>	|	EXEC	62	RET
+ 4		I-	i	E	<SG>			<SG>	|	EXEC	197	
										EXEC	207	RET
+ 5			t	E	<SG>			<SG>	|	EXEC	198	
										EXEC	207	RET
+ 6	E	IS	NOT	E	<SG>			<SG>	|	EXEC	108	RET
+ 7	E	IS	ALSO	Ξ	<SG>			<SG>	|	EXEC	109	RET
+ 8		E	is	E	<SG>			<SG>	|	EXEC	176	RET
H19		E	INST	E	<SG>		Ξ	<SG>	|	EXEC	85	COM
HA1		£	CLSO	E	<SG>		Ξ	<SG>	|	EXEC	77	COM
+ 1			CLSO	E	<SG>		E	<SG>	|	EXEC	80	COM
H20		E	v	E	<SG>		Ξ	<SG>	|	EXEC	105	
										EXEC	114	COM
H22		E	A	E	<SG>		E	<SG>	|	EXEC	105	
										EXEC	115 -	COM
H24			**	E	<SG>		Ξ	<SG>	|	EXEC	116	COM
H26		E	<	E	<SG>		E	<SG>	|	EXEC	100	
										EXEC	117	COM
+ 1		E	>	E	<SG>		Ξ	<SG>	|	EXEC	100	
										EXEC	118	COM
+ 2		E	NL	E	<SG>		E	<SG>	|	EXEC	10 0	
										EXEC	119	COM
+ 3		E	NG	E	<SG>	-»	E	<SG>	|	EXEC	100	
										EXEC	120	COM
+ 4		E	*	E	<SG>	•*	Ξ	<SG>	|	EXEC	100	
										EXEC	121	COM
+ 5		E	a-	E	<SG>	•»	E	<SG>	|	EXEC	187	
										EXEC	122	COM
H28		E	+	E	<SG>	->	Ξ	<SG>	|	EXEC	100	
										EXEC	123	COM
+ 1		E		E	<SG>	•4	E	<SG>	|	EXEC	100	

```
                                                                       EXEC 124      COM
   H30      E     *     E   <SG> | →      E     <SG> |                 EXEC 100
                                                                       EXEC 125      COM
    +1      E     /     E   <SG> | →      E     <SG> |                 EXEC 100
                                                                       EXEC 126      COM
   H32            NG*   E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 127      COM
   H34      E     ↑     E   <SG> | →      E     <SG> |                 EXEC 100
                                                                       EXEC 128      COM
   H36            SIGN  E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 129      COM
    +1            ENTI  E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 130      COM
    +2            ARCT  E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 131      COM
    +3            SQRT  E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 132      COM
    +4            EXP   E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 133      COM
    +5            LN    E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 134      COM
    +6            COS   E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 135      COM
    +7            SIN   E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 136      COM
    +8            ABS   E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 137      COM
    +9            ↓     E   <SG> | →      E     <SG> |                 EXEC 107
                                                                       EXEC 138      COM
   H38      E     L:    E   <SG> | →      E     <SG> |                 EXEC 87       COM
    +1                      <SG> |                   |                 RETURN
```

```
H39                      <OS> |                 |     RETURN
 +1 THE  E    OF    E    <SG> | →     E    <SG> |     EXEC 106     VR1
 +2      SL   OF    E    <SG> | →     E    <SG> |     EXEC 63      SL3
 +3      ELEM OF    E    <SG> | →     E    <SG> |     EXEC 213     COM
 +4      ATTR OF    E    <SG> | →     E    <SG> |     EXEC 214     COM
 +5      ↓    ↓     E    <SG> | → ↓   E    <SG> |     EXEC 197     COM
                                                      EXEC 207     RET
 +6      ↑    ↑     E    <SG> | → ↑   E    <SG> |     EXEC 198     COM
                                                      EXEC 207     RET
 +7           $     E    <SG> | →     E    <SG> |     EXEC 184     COM
 +8                      <SG> |                 |     RETURN

       PRODUCTIONS FOR EVAL
EV1                 I    [    | →     E    [    |                  E2
 +1           EVAL  I    <SG> | →     E    <SG> |     EXEC 7
                                                      EXEC 70      E2A
EV4                 (    I    | →     EV(  I    |     EXEC 71      E1
 +1                 (    [    |                 |                  *E1
 +2                      <SG> |                 |     ERROR 200    00
EV2      EVAL (     E    )    | →          FV   |     EXEC 64
                                                      EXEC 64      *E1
 +1 FV   E    (     E    )    | →          E    |     EXEC 72      *E2A
 +2                      <SG> |                 |     ERROR 201    00
```

UTILITY ROUTINES FOR ERROR RECOVERY

```
    QS0 UNSTACKS CHARACTERS UNTIL |→ APPEARS AT THE TOP OF THE STACK.
QS0                   |→    |                 |     RETURN
 +1                   <SG> | →                |                 QS0

    Q00: PROGRAM DOES NOT START WITH 'BEGIN'.
                                                 Q0                D1

    Q01: ILLEGAL FIRST CHARACTER OF A STATEMENT.
Q1               |→   <DC> |                  |     SUBR DEC    S1
 +1                   <SG> |                  |     SUBR QS0    *S1

    Q02: STATEMENT STARTS WITH ID NOT FOLLOWED BY A LEGAL CHARACTER.
Q2                    I    | →                |                 *Q2A
 +1                   ,    |                  |                 Q2A
 +2                   <OP> | →           ←←   |                 *E1
 +3                   <SG> |                  |     SUBR QS4    S1

    Q03: IN AN EXPRESSION: AN OPERAND WAS EXPECTED AND WAS NOT FOUND.
                                                 Q3                E2

    Q04: A BINARY OPERATOR WAS EXPECTED AND NOT FOUND.
Q4                    I    |                  |                 Q4A
 +1                   (    | →           E(   |                 Q4B
 +2                   [    |                  |                 Q4B
 +3                   :    | →                |     SUBR QS0    *S1
 +4                   FOR  |                  |                 Q98
 +5                   GO   |                  |                 Q98
 +6                   BEGI |                  |                 Q98
 +7                   <SG> | →                |                 *E1
Q4A                   I    | →            *   |     EXEC 7      *E1
Q4B                   <SG> | →       *   <SG> |                 *E1
                                                 Q5    SUBR QS0    *S1
                                                 QS1               Q5
                                                 QS2               Q5
                                                 QS3               Q5
                                                 QS4               Q5
                                                 Q7                Q5
                                                 Q8                Q5
                                                 Q9                Q5
                                                 Q11               Q5
                                                 Q12               Q5
                                                 Q13               Q5
                                                 Q14               Q5
                                                 Q17               Q5
                                                 Q18               Q5
                                                 Q19               Q5
                                                 Q20               Q5
                                                 Q21               Q5
                                                 Q22               Q5
                                                 Q25               Q5
                                                 Q38               Q5
```

```
                                                      Q39                 Q5
                                                      Q42                 Q5
                                                      Q6                  E2
                                                      Q10                 Q1
                                                      Q15                 E2
                                                      Q16                 E2
                                                      Q24                 Q99
                                                      Q41                 E1
                                                      Q44                 Q99
                                                      COL  EXEC 11        *E1
                                                      CAL  EXEC 11        *E1
                                                      Q2A                 Q0
                                                      CL1                 Q
                                                      CL2                 Q
                                                      Q    SUBR END
                                                           SUBR FLT       S1
                                                      HAL  HALT           IMP
                                                      IMP  ERROR 999      HAL
                                                      QDC  SUBR END
                                                           SUBR FLT       *CN
                                                      QSP  SUBR END       PID
PID                   P-ID  |                 |                           *SP
 +1                   |→    |                 |                           *CN
 +2                   <SG>  |  →              |                           PID
END                   ;     |  →              |        RETURN             IMP
 +1                   END   |  →              |        RETURN
 +2                   ELSE  |  →              |        RETURN
 +3                   <SG>  |  →              |                           *EN
FLT                   |→    |                 |                           *RE
 +1                   <SG>  |  →              |                           FLT

   Q98: IMPOSSIBLE ERROR AT S1.  ('|→' NOT IN STACK)
Q98              |→   <SG>  |                 |                            S1
 +1                   <SG>  |                 |        NSTK 2
                                                       STAK 0              Q9

   Q99: IMPOSSIBLE ERROR.  →→PANIC←←
                                                      Q99  SUBR QS0       *S1
```

APPENDIX II

TABLE 2: CHARACTERS AND HIERARCHIES

101	FOR	2	FOR	344
102	DO		DO	345
103	STEP		STEP	346
104	OWN		OWN	347
105	WHILE		WHIL	350
106	UNTIL		UNTI	351
107	VALUE		VALU	352
108	BEGIN		BEGI	353
109	LABEL		LABE	354
110	BOOLEAN		BOOL	356
111	HALF		HALF	357
112	REAL		REAL	360
113	LOGIC		LOGI	361
114	INTEGER		INTE	363
115	STRING		STRI	364
116	FORM		FORM	365
117	DERV		DERV	366
118	ATOM		ATOM	367
119	THE		THE	370
120	IS		IS	371
121	NOT		NOT	372
122	ST		ST	373
123	ND		ND	374
124	RD		RD	375
125	ALSO		ALSO	376
126	TH		TH	377
127	EVAL		EVAL	400
128	OF		OF	401
129	RECU		RECU	402
130	SYMBOL		SYMB	403
131	SWITCH		SWIT	404
132	ARRAY		ARRA	405
133	PROCEDURE		PROC	407
134	PRINT		PRIN	410
135	INDEX		INDE	411
136	OPERATOR		OPER	413
137	COMM		COMM	414
138	PARALLEL		PARA	416
139	INSERT		INSE	417
140	DELETE		DELE	420
141	COPY		COPY	421
142	ALTER		ALTE	422
143	LET		LET	423
144	FIRST		FIRS	424
145	LAST		LAST	425
146	BETWEEN		BETW	427
147	ALL		ALL	430
148	HAS		HAS	431
149	TO		TO	432
150	IN		IN	433

TABLE 2: CHARACTERS AND HIERARCHIES

151	ELEMENTS	ELEM	435
152	ATTRIBUTES	ATTR	437
153	BEFORE	BEFO	440
154	AFTER	AFTE	441
155	AND	AND	442
156	SUBLIST	SUBL	444
157	NIL	NIL	445
158	CONT	CONT	446
159	DL	DL	447
160	TEXT	TEXT	450
161	AMONG	AMON	451
162	COUNT	COUN	452
163	EX1	EX1	453
164	EX2	EX2	454
165	EX3	EX3	455
166	EX4	EX4	456
167	EX5	EX5	457
168	INFI	INFI	460

LAST SPECIAL CHARACTER FOR PHAS

169	TRUE	TRUE	461
170	FALSE	FALS	462
171		D1	463
172		D2	464
173		D3	465

```
          TABLE 3:  META-VARIABLES

 M  <OP>   < > NG = ∨ ≠ + - * / ↑ ∧  NL → INST CONT                                   0
 M  <TP>  REAL INTE BOOL LOGI HALF STRI FORM SYMB SUBL ATOM TEXT                      0
 M  <SP>  REAL INTE BOOL LOGI ARRA PROC HALF SWIT LABE STRI FORM SYMB                 0
 M  <UN>  ABS SIN COS LN EXP SQRT ARCT ENTI SIGN                                      0
 M  <DC>  <SP> OWN                                                                    0
 M  <ST>  ← ←← := :== .←                                                              0
 M  <RE>  = ≠ < > NL NG                                                               0
 M  <OT>  ELSE , ) ] ←] IF THEN E( ( [ ]→ ; END STEP UNTI WHIL DO :                   0
 M  <BI>  TRUE FALS INFI                                                              0
 M  <PN>  INST CONT                                                                   0
 M  <AT>  INDE OPER COMM                                                              0
 M  <BK>  .( EV( /[                                                                   0
 M  <OS>  ST ND RD TH                                                                 0
 M  <SM>  OF ST ND RD TH IS HAS TO IN B AND BEFO AFTE (                               0
 M  <PE>  OF AND , )                                                                  0
 M  <BA>  BEFO AFTE                                                                   0
 M  <SL>  FIRS LAST ALL BETW ( I                                                      0
 M  <EA>  ELEM ATTR                                                                   0
 M  <,]>  , ]                                                                         0


          THE TABLE AS LOADED

 <OP> <    >    NG   =    ∨    ≠    +    -    *    /    ↑    ∧    NL   →    INST C
  TP> REAL INTE BOOL LOGI HALF STRI FORM SYMB SUBL ATOM TEXT
 <SP> REAL INTE BOOL LOGI ARRA PROC HALF SWIT LABE STRI FORM SYMB
 <UN> ABS  SIN  COS  LN   EXP  SQRT ARCT ENTI SIGN
 <DC> REAL INTE BOOL LOGI ARRA PROC HALF SWIT LABE STRI FORM SYMB OWN
 <ST> ←    ←←   :=   :==  .←
 <RE> =    ≠    <    >    NL   NG
 <OT> ELSE ,    )    ]    ←]   IF   THEN E(   (    [    ]→   ;    END  STEP UNTI W
 <BI> TRUE FALS INFI
 <PN> INST CONT
 <AT> INDE OPER COMM
 <BK> .(   EV(  /[
 <OS> ST   ND   RD   TH
 <SM> OF   ST   ND   RD   TH   IS   HAS  TO   IN   B    AND  BEFO AFTE (
 <PE> OF   AND  ,    )
 <BA> BEFO AFTE
 <SL> FIRS LAST ALL  BETW (    I
 <EA> ELEM ATTR
 <,]> ,    ]
                    TABLE 3 LOADED CORRECTLY
```

```
•AND* HhCOHU SOURCE     14128151     08 DEC 63        Q OPER. t HJ02

                         00100130

23 SN        DUMP
   -BEGIN-TABLE.....................-                 .
                   LA311Q0»4J,                       | POSSIBLY  FOR  LABELS
33.................CRA0Lei«520,2J#.                  » FOR  CHAINING  LABELS,PROGS,ETC.,
 3                 FPT12Q.3),                        I FORMAL  PARAMETER  TABLE
J3_________________SVMdt.400.-4J_____________________|_GENERAL  SYMBOL__TABLE____________
63    I      DATA
.  .  .  .  .  ___BOOLEAN,.... INTEGER,  SINGLE.  DOUBLE,  _1__________________
                   LOGICAL* FUNCTION. SUBLIST. LABEL,
  _______--________FORMULA.  TEXT,       TRUMP,    STRING,_______________________
                         SYMBOL.  THOUGHT.  CLASS,    LAST.
                  .ANY_____.__MODEl.......MODE2,___M0DE3       -
      )      CELL
 . . . . . . . .  MAX*T2#                   | MAXIMUM FIXED STORAGE AND__MINIMUM-T.EMP.
   INTE86R * STEPPE(STEPPE)  »          I TYPICAL  STEP  SIZE
   8R.....',..…-14377  «  THE,  8R447  «  TRU.______________________:________
   8R       11263  «X20/X21/X22/X23/X24/X25/X26/X27/X28/X29/
                         X3U/X31/X32/X33/X34/X35/X36/X37/X38/X39V
  _______________________X40/X41/X42/X43/X44/X45/X46/X47/X48/X49/_______________
                  X50/X51/X52/X53/X54/X55/X56/X&7/X58/X59/.......
                  X60/X61/X62/X63/X64/X65/X66/X67/X68/X69/
                  X70/X71/X72/X73/X74/X75/X76/X77/X78/X79, -.....
   8R       56441. » X100/X101/X102/X103/X104/X105/X106/X107/X108/X109/
  __________-_________X110/X111/X112/X113/X114/X115/X116/X117/X118/X119/—
                      X120/X121/X122/X123/X124/X125/X126/X127/X128/X129/
  ____________________X130/X131/X132/X133/X134/X135/X136/X137/X138/X139/-
                      X140/X141/X142/X143yxi44/X145/X146/X147/Xl48/X149/
  ____________________X150/X151/X152/X153/X154/X155/X156/X157/X158/X159/-
                      X160/X161/X162/X163/X164/X165/X166/X167/X168/X169/
  __________=_________X170/X171/X172/X173/X174/X175/Xi76/Xl77/-X178/X179/-
                      Xl80/X181/X182/X183/Xl84/xia5/X186/X187/X188/X189/
  ._......:___________X190/X191/X192/X193/X194/X195/X196/X197/X198/X199/-
                      X200/X201/X202/X203/X204/X205/X206/X207/X208/X209/
  . . . . . . . . . .-.....X210/X211/X212/X213/X214/X215/X216/X217/X218/X219/-
                      X220/X221/X222/X223/X224/X225/X226/X227/X228/X229/
  ____________________X23Q/X231/X232/X233/X234/x235/X236/X237/X238/X239/_
                      X240/X241/X242/X243/X244/X245/X246/X247/X248/X249/
  . . . . . . . .1......-  X250/X251/X252/X253/X254/X255/X256/X257/X258/X259/-
                      X260/X261/X262/X263/X264/X265/X266/X267/X268/X269/
  ....________________X270/X271/X172/X273/X274/X275/X276/X277/X278/X279/—
                      X280/X281/X282/X283/X284/X285/X286/X287/X288/X289/
   ____________________X290/X291/X292/.X2-9.3/X294/X295/X296/_X29Z>LX29&/X2A9.J-
   8R       14300  .•X80/PAR/X82/TAR/X84/X85/X86/RAG/X88/X89/                ""0
    _........_______________/ERROR/LBS/UBH.  -I UNDEF LABL EXIT, LB-STORAGE.-UB-HISTOFJO
   8R       11652 »V59/    /    /V60/    /    /V58/    /    /V61.
   8R_______-63224   TbMP,_____________________I TEMP BIT T10$26__________:___________
   8R       63262   R*LB / RELA / CXT ,     I RELATIVE  ADDRESSING  PARAMETERS
        -VAL2.8STAALKA.-T-1,F0RV,
   8R    10 0 00 =' iNCON  ,               | MODE 0 INTEGER  CONSTANT
```

```
8R      00 1 00 = ABVAR ,               | ABCON  OR  FIXED  VARIABLE
8R      00 2 00 = ARRAY ,               | BLOCK  ZERO  ARRAY
8R      00 3 00 = CODEP ,               | CODEPIECE
8R      00 4 00 = LABLE ,               | LABEL
8R      10 5 00 = THUNK ,               | ANOTHER  PARAMETER
8R      00 6 00 = PRCDR ,               | PROCEDURE
8R      00 7 00 = SWTCH ,               |  SWITCH
8R      10 7 00 = TMASK ,               | FOR  EXTRACTING  CLASS
8R      77 7 77 = X7,                   | ADDRESS  EXTRACTOR
8R  1 00 0 00 = SHIFT = MODE0 ,         | LEFT  SHIFT 15  AND  MODE 0  BIT
8R  2 00 0 00 = SECND ,                 | FIRST  PLACE  PARAMETER
8F      1 . -5 = R15,                   | RIGHT  SHIFT  15
8R            70-+ CODSTK ,             | CODELOC  AS  A  LOGICAL  STACK
8R      56 00001 = STORAGE ,            | TO  GET  VALUE  OF  FUNCTION
8R      57 00000 = R0 ,                 | POINTS  TO  VALUE OF PARAMETER
8R      77 00000 = HEIGHT ,             | LOCAL  CONTEXT  REGISTER
8R 20000 00057 = RZ ,                   | HOLDS LOC ( ANSWER ) TO FORMAL PARAMETE
      8R11670 + L18,               | MAX STORLOC
8R          10135 + Q1/  /  /  /NFALTS,| N45 = COLUMN SWITCH, N49 = NO. OF SEM.
8R          11702 + E17 ,              | FOR  FINDING  WORD  TO  ZERO
8R          40106 + EXIT ,              | EXIT  FROM  A  SENTENCE
8R          40144 + STCM ,              | STORE  COMMAND
8R          40170 + SAFEN ,             | L20  SAFENS  THE  ACCUMULATOR
8R          16474 + DUMPWIDTH ,        | INITIALLY   LXP  0   5,R2
8L0012 57        = LXPR0 ,              | USED  IN  CONSTRUCTING  LXP  0   N,R0
8L0012 62        = LXPR2 ,             | LATER         LXP  0   <Q1>,R2
8R      65501 + RUDY,           |   RUDY IS A VARIABLE ...
8R          63500 = VCP    ,            | VALUE  OF  CODEPIECE
8R          63511 = CHEND  ,            | CHAIN  END
            CLUTCH,                     | CONTROL  LEVER
            SWCUNT,                | COUNTER FOR SWITCHES
      A, B,  8R11666+C,            | TEMPS FOR ATLAS, C IS USED BY +100+ AND OTHE
            T,TT,TTT,IV,                | VOLATILE  TEMPS
            FND,                        | FORMAL  PARAMETER
            ID,F,                       | FUNC. DESIG. , ITS  LOCATION
      8R45 = C1,         | CHANGE IF MORE SYMBOLS ARE ADDED
      8R11667+TYPE, KEY, RELOC, |COLS 2,3,4 IN SYMB
      X,
      8R11721+X1,
      8R11722+X2,
       X3,X4, Y1,Y2,Y3,Y4,
      Q28, Q112,
      ATTRIBUTE,                 |  SWITCH FOR ATTRIBUTE OR VALUE
      EVAL1,                     |  TOP OF STACK EVAL
            CLASS,                      | INCON,ABVAR,...,SWTCH
            LH,RH,                      | LEFT HALF,RIGHT HALF
            LEV,                        | COMPILE TIME BLOCK LEVEL
            CRADLOC,                    | VERY  BOTTOM  OF  CRADLE
            CSS,                        | ADDRESS  OF  CURRENT  STORAGE  SETTER
            SYMBO                       | LOC OF 1ST LINE IN SYMB
;     STACK
            LSS ,                       | ADDRESS  LAST  STORAGE  SETTER
            LADLE,                      | FOR  DISHING  OUT  LABELS
            ACT ,                       | THUNK  STACK
            BASE ,                      | RIGHTMOST  RELOCATION  BASE
```

```
67         EVAL,                      | STACK OF HEAD OF EVAL CHAINS
70              SWICH, RET,
72              STAB                  | FOR  LABELING  SYMB
    ;      RSTAK                      | RUN-TIME  STACKS
73              MAIN,                 | PAIRS FOR PROCEDURES,TRIPLETS FOR THNK
75              NEXT,                 | NEXT  AVAILABLE  RUN-TIME  LOCATION
77         R0,R1,R2,R3,R4,R5,R6,R7,L1 | INDEX REGISTERS FOR R.TIME R.
17  ;      TITLE
21              REAL,INTE,BOOL,       | TYPES OF VARIABLES
                LOGI ,HALF,LIST,FORM, STRI,
                MARK,                 | MARKS  LABELS AND PROCEDURE IN SYMB
                LABL,SWIT             | DESIGNATIONAL  EXPRESSIONS
    ;   0+
           X+8R11237;
23         X1+(X+1) v MODE1 v FORMULA;
30         X2+ ACC + 1;
32         X3+ ACC + 1;
34         Y1+(X+10) v MODE1 v DOUBLE;
41         Y2+ ACC + 2;
43         Y3 + ACC + 2;
45         Y4 + ACC + 2;
47         Q1 + FALSE ;            | ZERO  N45
51         Q28 + XEQ 28;
53         Q112 + XEQ 112;
55

    | COMPILE-TIME  INITIALIZATION
           PUSH(SWICH,0);
60         SYMB0+LOC(SYMB);                  | BASE  OF  SYMB  TABLE
62         CXT+0;RELA+ACC;RELB+ACC;          | ZERO  ALL  RELATIVE  THINGS
66         BASE  +  0  ;                     | EVEN  THIS  ONE
70         RUDY + ACC;               | RUDY + FALSE
71         ATTRIBUTE + ACC;           |  ATTRIBUTE + FALSE
72         MAX + STORLOC ;                   | END  OF  FIXED  STORAGE
74         CRADLOC + LOC(CRADLE)-320 ;       | WHERE  TO  START  ON  CRADLE
77         LEV+8R100000;
01         T+LOC(SYMB)v8L2;
04         ENTER(SYMB;T,T,T,T);
16         CODELOC + CODELOC - 11 ;       | SKIP  THOSE  SILLY  LXP'S
21         CODE(MARKJUMP(<X115>));          |INITIALIZE CHANI ACC
25         CLUTCH + FALSE                  | DON'T START WITH CLUTCH  IN
    +  1+
30         LEV+LEV+8R100000;
33         PUSH(STAB, LOC(LAB));
36         PUSH(STAB,STORLOC);
41         PUSH(STAB,(LOC(SYMB)v8L2));     | SCATTER  LABEL
45         PUSH(LADLE,0);                 | MARK  BLOCK  WRT  LABELS,PROCS,...
50         CLUTCH + FALSE ;               | CONTROL  IS  HERE
52         CXT+ CODE(MARKJUMP(X85));       | BLOCK  ENTRY  ROUTINE
60              CODSTK+(<CSS>^X7)*SHIFT;   | NEW  HEAD  OF  CHAIN
64              <CSS>+(<CSS>^-X7)+CODELOC; | CHANGE  ELEMENT  OF  SUPER CHAIN
70              PUSH(LSS,CSS);CSS+CODELOC; | DOWN  ONE  LEVEL
75              TALLY(CODELOC) ; CODE(MARKJUMP(<X46>)) $ ;
05         JUMP(EXIT) ;                   | END  OF  EXEC  1
10         'ENEX'
```

```
3           TEST[LEFT2, CLASS] ∨
0           TEST[LEFT2, SYMBOL] →
2           CODE(MARKJUMP[<X187>])!
0           MARKJUMP[DATATERM] $                   | X1← EXP
2      ↓ 97↓
3           TEST[LEFT3, SYMBOL] →
3           SET[RIGHT2, SYMBOL];
7           CODE(MARKJUMP[<X200>])                 | CONTENTS
3           ! FAULT 97 $
0      ↓103↓
1           TEST[LEFT4, SYMBOL] →
1           CODE(MARKJUMP[<X201>])                 | LOCAL DUSRIPTION LIST
5           ! FAULT 103 $
2      ↓104↓
3           CODE(MARKJUMP[<X121>])              | RECOVER PHANTOM
7      ↓106↓
0           TEST[LEFT4, SYMBOL] →
0           TEST[LEFT2, SYMBOL] →
0           SET[RIGHT2, SYMBOL];
4           CODE(MARKJUMP[<X127>])              | VALUE RETRIEVAL
0           ! FAULT 106 $
5           ! FAULT 108 $
2      ↓108↓
3           TEST[LEFT5, SYMBOL] →
3           MARKJUMP[DT];
4           CODE(MARKJUMP[<X139>])              | IS NOT
0           ! FAULT 109 $
5      ↓110↓
6           CODE(MARKJUMP[<X157>])              | DESCRIPTION LIST STORE
2      ↓109↓
3           TEST[LEFT5, SYMBOL] →
3           MARKJUMP[DT];
4           CODE(MARKJUMP[<X134>])              | IS ALSO
0           !FAULT 109 $
5      ↓ 66↓
6           TEST[RIGHT2,SYMBOL] →
6           POP[BASE, RELA];
1           CODE(X1←RIGHT2;                     | X1←ADHCH
0           MARKJUMP[<X150>]);                  | UNITE SYMBOL BITS
4           MARKJUMP[<X136>]) $                 | STACK UNCARRIED
0      ↓ 67↓
1           RIGHT1←1
       ↓ 46↓
4           RIGHT1←0
       ↓ 48↓
7           CODE(MARKJUMP[<X169>])                | ALL EXP
3      ↓ 53↓
4           CODE(X1←LEFT2;
3           X2←LEFT4;
2           MARKJUMP[<X165>]);                    | OSE BEFORE EP
6           VALUE2←X1+0)
0      ↓ 54↓
7           CODE(X1←LEFT2;
6           X2←LEFT4;
5           MARKJUMP[<X166>]);                    | OSE AFTER EP
```

```
           T+ABVAR; MARKJUMP(DECLARE); JUMP(EXIT);    |VARIABLE LIST
       'ALST'CODE(MARKJUMP(FLAD1);STORLOC+X1); |
           MARKJUMP(V60); MARKJUMP(DECLARE); JUMP(EXIT); |ARRAY LIST
5336   'FLST'ENTER(FPT;LEFT1,FNO,FALSE) ;  | FORMAL  PARAMETER  LIST
5346       FNO + FNO + 1 ; JUMP(EXIT) ;    | COUNT  THE  PARAMETERS
5354   'VLST'FPT(LEFT1,,$)  +  TRUE  ;     | VALUE  LIST
5363       -SIGNAL+ FAULT 5 $;JUMP(EXIT);| IT  ISN'T  THERE
5373   'SLST'FNO+FPT(LEFT1,$,);-SIGNAL+FAULT 6; | SPECIFIER  LIST
5411       FPT(0,,$) + T+ABVAR; MARKJUMP(DECLARE);    |CALLED BY VALUE,DECLARED I
5424             CODSTK+TAR;TALLY(CODELOC); | EVALUATE  IT
5427             CODSTK+(THUNK+FNO)*SHIFT+CXT;TALLY(CODELOC);|
5435             LEFT4+LEFT1;LEFT1+0;       |     ID    +    RZ
5441             RIGHT2 + TYPE + RZ ;       | LOC ( VALUE )  IS  IN  R0
5444             JUMP(STORE) ;
5447       ENTER(SYMB;LEFT1,TYPE+THUNK,FNO,CXT)$$ | CALL  BY  NAME
5463     + 7+
5464       PUSH (BASE,0);                   | A  NEW  BASE
5467       CONST(LEFT2) + TEST(LEFT2,BOOLEAN) + LEFT2+LEFT2vLOGICAL$ ;
5513       MARKJUMP(FIND);
5514 SN    COR           0400000002        OAD  0   2) JUMP  TO  RIGHT  PLACE
5515   'FOREVER'  JUMP(FOREVER) ;           |
5516             JUMP(F7);                  |-1  - IN  SYMB
5517             JUMP(F7);                  | 0  FUNCTIONLESS  PROCEDURE
5520             JUMP(VARIABLE);            | 1  VARIABLE---FIXED  OR  DYNAMIC
5521             JUMP(F7);                  | 2  ALL  ARRAY  CASES  ELSEWHERE
5522             JUMP(F7);                  | 3  CODEPIECE - ONLY  IN  THUNKS
55               JUMP(DESL);                | 4  LABEL  IN  COND'L IN CODEPIECE
55               JUMP(FPAR);                | 5  FORMAL  PARAMETER
5525             JUMP(FUNC);                | 6  FUNCTION
5526             JUMP(F7)  ;                | 7  SWITCH ---  SAME  AS  ARRAYS
5527 SN    COR            1604
5530   'F7'  FAULT  7 $ ; JUMP(EXIT) ;      | EXIT  AFTER  CONSTANT  OR  FAULT
5536
     'VARIABLE'
           RIGHT2+ KEY+MODE1+TYPE+TEMP;   | THE  CORE  OF  THE  EXPRESSION
5543       BASE  +  RELOC                 | SET  IT'S  RELOCATION  BASE
       + 21+
5546   'FUNC' MARKJUMP(SAFEN);MARKJUMP(CALL); | RACC ALREADY SAFE IN  EXEC  21
5550   'FRET'ACC + STORAGE ;             | FUNCTION VALUE IS IN 1,R-1
5551   'GET' TTT+ACC;TT+TYPE+MODE1;      | THE  CORRECTION  AND  THE  EXPRESSI
5555       T    +    CODELOC   ;         | WHERE  THE  CORECTION  WILL  BE
5557       CODE(ACC+TT;VALUE2+ACC);      | GET  IT  INTO  THE  ACC
5563       RIGHT2 + RIGHT2 v TYPE ;      | IT  NEEDS  TO  BELONG
5566       <T> + <T> + TTT ; BASE + CXT ;| ALTER  THE  ACCESS
5573       JUMP(EXIT);                   | -30-
5576   'FPAR'
           MARKJUMP(SAFEN);              | SAFEN  THE  ACCUMULATOR
5577       CODSTK+TAR;TALLY(CODELOC);    | TRM   V203
5602       CODSTK+(THUNK+KEY)*SHIFT+RELOC;TALLY(CODELOC);| V203'S  PARAMETER
5610       ACC+R0;JUMP(GET);             | THE  REST  PARALLELS  FUNC
5612   'DESL'FAULT 198
       + 92+
           CODE(MARKJUMP(<X100>))           |CONCATENATE
       + 96+
5622       POP(SWICH,0);
```

```
          VALUE2←X1+0)
    + 55+
          CODE(X1←LEFT2;
          X2←LEFT4;
          MARKJUMP(<X168>))           | BETW PO AND PO
    + 56+
          CODE(X1←LEFT2;
          MARKJUMP(<X174>))           | ALL BEFORE PO
    + 57+
          CODE(X1←LEFT2;
          MARKJUMP(<X175>))           | ALL AFTER PO
    + 58+
          CODE(MARKJUMP(<X180>))      | INSERT
    + 59+
          TEST(LEFT2, SYMBOL] →
          CODE(MARKJUMP(<X181>))      | DELETE
          ; FAULT 59 $
    + 62+
          TEST(LEFT4, SYMBOL] →
          MARKJUMP(DT);
          CODE(MARKJUMP(<X182>))      | ALTER
          ; FAULT 59 $
    + 69+
          LEFT1 ← LEFT1 ← 8R11;
          MARKJUMP(8R11771);
          CODELOC ← CODELOC ← 3;
          CODE(X1←X2);
          MARKJUMP(8R11655);
          CODE(                        | GET OPERATOR
          MARKJUMP(<X151>);            | DATA TERM BITS
          MARKJUMP(<X136>);            | STACK
          MARKJUMP(<X100>))            | CONCATENATE
    +78+
          TEST(LEFT2, SYMBOL] →
          CODE(MARKJUMP(<X207>))
          ; FAULT 78$                  |ERADL
    +79+
          'EX79'
          CODE(MARKJUMP(<X213>)))      | CREATE
          CLEAR(RIGHT2);
          SET(RIGHT2, SYMBOL]
    +90+
          CODE(ACC←1);
          JUMP(EX79]
    +166+
          CODE(X2←LEFT3));
          TEST(LEFT2, CLASS]→
          CODE(MARKJUMP(<X163>));
          CODE(MARKJUMP(<X161>))$;
          CODE(VALUE2←X1+0)
    + 63+
          TEST(LEFT2, SYMBOL] →
          SET(RIGHT2, SYMBOL];
          CODE(JUMP(RET));
          POP(RET,0);
```

```
        ASSIGN(FLAD2)
        ;FAULT 63 $
 ↓ 47↓
        TEST(LEFT2, SYMBOL) →
        CODE(MARKJUMP(<X205>));
        MARKJUMP(<X206>);
        VALUE1 ← ACC) ;
        TEST(LEFT2, INTEGER) →
        RIGHT1←LEFT2
        ; FAULT 47 $ $
 ↓173↓
        CLEAR(RIGHT2);
        SET(RIGHT2, SYMBOL);
        CODE(MARKJUMP(<X108>))        | CHAIN SUBLIST
 ↓175↓
        TEST(LEFT2, SYMBOL) →
        CODE(MARKJUMP(<X156>))        | ALPHA GETS ATTRIBUTE BITS
        ;FAULT 175 $
 ↓176↓
        TEST(LEFT4, SYMBOL) →
        CODE(MARKJUMP(<X101>))        | GENERAL STORE
        ; FAULT 176 $
 ↓177↓
        CODE(X2←LEFT4;
        MARKJUMP(<X164>));             | OSE INT
        VALUE2←X1+0)
 ↓178↓
        CODE(MARKJUMP(<X199>))          | UNCARRY IF UNIT INTERIOR
 ↓179↓
        TEST(RIGHT2, INTEGER) →
        POP(BASE, RELA);
        RIGHT2 ← RIGHT2 ← MODE1;
        CODE(X1←RIGHT2;
        MARKJUMP(<X151>); | MAKE INDEX A DATA TERM
        MARKJUMP(<X136>);
        MARKJUMP(<X100>)) ; FAULT 179 $
 ↓180↓
        TYPE ← SUBLIST
 ↓181↓
        CODE(X1←LEFT2;
        X2←LEFT3;
        MARKJUMP(<X162>));
        VALUE2← X1+0)
 ↓182↓
        SET(RIGHT1, SYMBOL);
        CODE(MARKJUMP(<X118>))          | STACK NIL
 ↓183↓
        TEST(LEFT4, SYMBOL) →
        'INST'
        CODE(MARKJUMP(<X204>));
        VALUE2 ← ACC);
        SET(RIGHT2, BOOLEAN)
        ; FAULT 183 $                   | E == E
 ↓184↓
        CLEAR(RIGHT2);
```

```
        SET[RIGHT2, SYMBOL];
        TEST[LEFT2, SYMBOL] →
        CODE(MARKJUMP[<X205>]]
        MARKJUMP[<X206>])]
        TEST[LEFT2, INTEGER] →
        CODE(ACC ← LEFT2)
         ; FAULT 184 $ $
        ; CODE(MARKJUMP[<X202>])
  +185+
        SET[RIGHT2, SYMBOL];
        CODE(ACC←0; MARKJUMP[<X202>])
  +186+
        CODE(MARKJUMP[<X203>])
  +187+
        TEST[LEFT4, SYMBOL] ∧ TEST[LEFT2, SYMBOL] →
        C←3 ; JUMP[EX100] $
  +188+
        SET[RIGHT1, SYMBOL];
        CODE(X1←0; MARKJUMP[<X136>])        | STACK CONT
  +189+
        TEST[LEFT2, FORMULA] →
        POP[BASE, RELA];
        COMT 7 ← LEFT2 + 1;
        CODE(X1←COMT 7;
        MARKJUMP[<X150>];
        MARKJUMP[<X136>])        |  DL(FORM)
        ; FAULT 189 $
  +191+
        CODE(X1←LEFT2;
        MARKJUMP[<X167>])                  | PO
  +192+
        SWICH←0;
        PUSH[FLAD2, 0];
        CODE(JUMP[FLAD2]);
        ASSIGN[FLAD3]
  +193+
        SWICH = 0 →
        PUSH[FLAD3, 0];
        CODE(X2←X2);
        CODELOC ← CODELOC -2;
        CODE(JUMP[FLAD3]);
        PUSH[RET, CODELOC];
        SWICH←1 $
  +194+
        TEST[LEFT2, CLASS]→CODE(MARKJUMP[<X215>]);
        MARKJUMP[DT]
        $
  +195+
        SET[RIGHT1, SYMBOL];
        CODE(X1←1; MARKJUMP[<X136>])       | STACK 1
  +196+
        CODE(X1←4; MARKJUMP[<X136>];       |STACK 4 NOOT 3
        MARKJUMP[<X156>])                  |STACK 3 AS AN ATTRIBUTE
  +197+
        CODE(MARKJUMP[<X188>])             | POP
```

```
    +198+
          CODE(MARKJUMP[<X189>])              | PUSH
    +199+
          CODE(MARKJUMP[<X210>])              | JUMP
    +200+
          CODE(X1+LEFT2;
          MARKJUMP[<X177>])                   | ALL RT
    +201+
          TYPE + TEXT
    +202+
          LEV + LEV
    +203+
          TEST[LEFT2, SYMBOL] +
          CODE(MARKJUMP[<X183>];              | COUNT
          MARKJUMP[<X122>]) ;
          CLEAR[RIGHT2];
          SET[RIGHT2, INTEGER]
          ; FAULT 203 $
    +204+
          CODE(X1+LEFT2;
          MARKJUMP[<X178>])                   | BEFORE PO OF
    +205+
          CODE(X1+LEFT2;
          MARKJUMP[<X179>])                   | AFTER PO OF
    +206+
          CODE(X2+LEFT2;
          MARKJUMP[<X160>];
          VALUE2+X1+0)                        | OSE
    +207+
          CODE(MARKJUMP[<X122>])              | DECREMENT CHAIN ACC
    +208+
          CODE(X1+2; MARKJUMP[<X136>];        | STACK OPER
          MARKJUMP[<X156>])                   | MAKE ATTRIBUTE
    +209+
          PUSH[SWICH,0]
    +210+
          XEQ 28 +XEQ 216;
          XEQ 112 + XEQ 215;
          ALFA + CODELOC;
          CODE(MARKJUMP[<X195>];
          MARKJUMP[FLAD3];
          JUMP[ALFA])
    +211+
          TEST[RIGHT2, SYMBOL] +
          CODE(MARKJUMP[<X128>]) $                    | POINT TO CONTENTS
    +212+
          TEST[LEFT2, SYMBOL] +
          CODE(MARKJUMP[<X211>]);             | MARK ALPHA FOR VARIALBE
          XEQ 112 + XEQ 215 $
    +213+
          TEST[LEFT2, SYMBOL] +
          XEQ 112 + XEQ 210;
          CODE(MARKJUMP[<X140>])                      | ELEM OF
          ;FAULT 213 $
    +214+
```

```
        TEST[LEFT2, SYMBOL] →
        XEQ 112 ← XEQ 210;
        CODE(MARKJUMP[<X196>])              | ATTR OF
        ; FAULT 213 $
 ↓215↓
        MARKJUMP[DT];
        CODE(MARKJUMP[<X101>]);
        CODE(MARKJUMP[<X121>])
 ↓216↓
        XEQ 28 ← Q28
 ↓217↓
        PUSH[FLAD3,0];
        ALFA ← CODELOC;
        CODE(MARKJUMP[<X194>]);
        MARKJUMP[FLAD3];
        JUMP[ALFA])
 ↓218↓
        SET[RIGHT1, SYMBOL]
 ↓219↓
        CODE(MARKJUMP[<X102>]);
        MARKJUMP[<X122>])              | DELETE SE
 ↓225↓
        SET[RIGHT1, CLASS]
 ↓226↓
        CODE(X1←7;
        MARKJUMP[<X136>];
        MARKJUMP[<X127>]);
        COMT 8 ← CODELOC + 5;
        CODE(X1←COMT 8;
        MARKJUMP[<X136>];
        MARKJUMP[<X101>]);
        PUSH[FLAD2, 0];
        CODE(JUMP[FLAD2]);
        TALLY[CODELOC];
        CODE(LEFT2←ACC)/
                                   00:01:41

00.0000
 ↓227↓
        CODE(JUMP[<COMT 8>]);
        ASSIGN[FLAD2]/
 ↓  8↓
        LEFT1 = TRU-1 → ACC ←FORMULA + X !
                         ACC ←TRU - LEFT1 + TRE $ ;
        RIGHT1 ← ACC + MODE1 + SECND
 ↓  9↓
        MARKJUMP[FIND];
        TYPE = SYMBOL →
        PUSH[BASE, 0];
        JUMP[VARIABLE]
        $
```

```
    + 10+
            MARKJUMP(CALL)                      | COMPILE A CALL OF A PROCEDURE
    + 11+
            MARKJUMP(SAFEN);                    | SAFEN THE ACCUMULATOR
            PUSH(ACT,8L1);                      | CAN'T MARK WITH ZERO (WHICH IS LEGAL)
            PUSH(FLAD4,0);CODE(JUMP(FLAD4));    | JUMP AROUND PARAMETERS
            JUMP(NEWTH)                         | GET READY FOR FIRST PARAMETER
    + 12+
            CONST(LEFT2)+KEY+LEFT2^8R77777;     | GET ADDRESS OF CONSTANT
               (LEFT2^MODE0)=0+ACC+ABVAR;       | CONSTANT WET THE POOL
               ACC   +    0    $    ;           | MODE 0 ABCONS HAVE NO CLASS AT AL
               MARKJUMP(FIND);                  | LOOK UP ID ^ DISSECT ENTRY
 SN         COR          0400000002               OAD  0  2; PICK OUT THE RIGHT CL
2   'RAVEN'    JUMP(RAVEN);                     |     FOREVERMORE
               JUMP(F18);                       | -1  - IN SYMB
               JUMP(PRSDR);                     | 0   FUNCTIONLESS PROCEDURE
               JUMP(WARBLE);                    | 1   FIXED OR DYNAMIC VARIABLE
               JUMP(ARAY);                      | 2   ARRAY
               JUMP(F18);                       | 3   SYNTAX DISCRIMINATES CODEPIECES
               JUMP(FLAB);                      | 4   ACTUALLY A LABEL
               JUMP(THNK);                      | 5   FORMAL PARAMETER
               JUMP(PRSDR);                     | 6   FUNCTION   DESIGNATOR
               JUMP(SVITCH);                    | 7   SWITCH
 SN         COR          1604              OCA  0    900
3   'F18'      FAULT 12; JUMP(EXIT);            | SOME KIND OF ERROR IN AN ACT, PAR
    'THNK'     T+KEY;KEY+RELOC;ACC+T+THUNK;JUMP(FINE);| 105  PROC,FNO
    'PRSDR'    KEY+RELOC;ACC+PRCDR;JUMP(FINE);         | 006  PROC
    'SVITCH'   KEY+RELOC;ACC+SWTCH;JUMP(FINE);         | 007  SWIT
    'FLAB'     ACC + RELOC + LABLE ;                   | 004  DEST,LEV
    'ARAY'                                             | 002  START OR 2 BASE
    'WARBLE'   RELOC+T+KEY;KEY+RELOC;ACC+SHIFT+T;      | 2NNN BASE,KEY
               ACC + ABVAR * ACC $ $ ;                 | 001  LOCATION
    'FINE'     ACT + ACC * SHIFT + KEY ;| USE THIS THUNK INSTEAD
    'NEWTH'    PUSH(ACT,8L0003+CODELOC) | INITIALIZE NEXT THUNK NOW
 CO + 13+ NOT  YET
    | 15+  SEE DESIGNATIONAL EXPRESSIONS
    | 16+  SEE ARRAYS
    | 17+  SEE ARRAYS
    + 18   TITLE  EQUALS ;
      24+
            TT  +  (  LEFT2^LAST  ) + VCP;| ACCUMULATOR TEMP
            CODE(TT+LEFT2);               | SAVE THE VALUE OF THE EXPRESSION
            CODSTK+LXPR0+VCP;TALLY(CODELOC);| LXP 0  LOC,R0
            CODE(JUMP(X84));JUMP(NEWTH)   | THUNK ALREADY SET FOR CODEPIECE
    + 20+
            MINUS(CODELOC) ;                    | GET SET FOR POP
    'UNLOAD'   POP(ACT,0);                      | DELETE PHANTOM CODEPIECE
            PUSH(CODSTK,ACT) ;                  | MOVE THUNK
            ACT ≠ 8L1 + JUMP(UNLOAD) $ ;        | 8L1 IS THE MARKER
            TYPE+SYMB(LEFT2,$,,)^¬TMASK;        | RETRIEVE THE OF FUNCTION
            - SIGNAL + FAULT  20  $    ;        | A FUNCTION MUST BE THERE
            POP(ACT,0);ASSIGN(FLAD4)            | CODEPIECES AND THUNKS DONE, START CALL
 C  + 21+ LOOK  IN  EXEC  7
    + 22+
            LEFT2 + LEFT1;
```

```
        MARKJUMPIFIND];
        ACC*1 •»•.
        RIGHT1 *• KEY + MODEI • TYPE • TEMPI
        BASE    RtlQO I FAULT 22 $       -
* 23+
        RIGHT2 <• RIGHTZ-A-"«<8R6332I>        1- SET IR IQHT2,-MODEOX
| 25+     SEE ARRAYS
* 26*
        PUSH(FLAOI.O);
        SWCONT ⁃ LfcFT2J          I SAVE LEFTF2, SWCONT-IS-NOI-IN-USE-NOW-
        LEFT2 * VAL2J
        MARKJUMP[FIND];
        T - KEY   MOUE1 • TYPE • TEMP;
        RELA «• CXT;
        RELB «- BA,E;
        LEFT2 «- SWCONT*-.........I RESTORE LEFT2
        CODEC T*LEFT2 > 0 * JUMP(FLADI) $ > 1
        CODEC MARKJUMPIFLAD31~J MARKJUMP(ALFA1 )1
        MARKJUMP( INCR'EI I
        COOE(JUM?|BETA) );•..
        ASSIGNtFLAD11
*• 27+
        pusHiFuAUi.u);
      -  TES7UEFT2,BOOLEAN! - FAULT-27 S)
        CODE(LEFT2 •* MARKJUMPIFLAD31J JUMPtAL^AH
                    JUMPtFLADIJS);
—       *-28* ASSIGNIFLAQ1J

        CODEC MARKJUMPIFLAD31)

•     _*_.30*  RIGHT2.*-FQ«V: ALFA*CODELOC
        TESTIL5FT2#BOOLEANJ v TEST{LEFT2.TRUMPI *
1       TESTIL5FT2.TKUMP) •»
        MARKJUMP(8R11765]; LEFT2 «• <8R63226>|
        MARKJUMP-t<X57>.]_.$ ; PUSHIFLAD1,01 ;....
        CODEC -v LEFT2 *• JUMP(FLAD1]$)I FAULT 30 $
+-31+
        »EXE31«
.........HUSH[FLAD2,oM CODE< JUMPIFLAD?11; ASSI^NIFLADIJ
+ 32*
        P0PJ FLAD4,T11J_CODEIJUMPt<T1>11;.....ASSIGNIFLADAJ
* 33*
—.:. ASSIGN[FuADIJ........ . . . . . . . ......... — . . . . . . . . . . . . . . . .
* 34*
........ ASSIGNIFLAD2]. . . . . . . . . . . . . . . . . . . .
* 35+
            MARKJUMPtSASS     U........| -ASSIGN..... SIZES_0F____IN NER_BLOCJ
        ENTER[SYMB:STA8JiPOP.JSTAB»0JJ I ENTER SCATTER LABEL'
        P0PISTA3.ST0RL0C1J.................1 PREVIOUS VALUE OF..STORLOC
        CODE{JUMP{X82]> |                  | END OF THE PROCEDURE BODY
   .........MAHKJUMPtATLAsl i-...................| ASSIGN SOME-—STUFF;
        CXT <- RIGHT! :                    I POP UP OLD CONTEXT
        CLUTCH <- TKUE___: . . . . . . . . . . J NO -CONTROL—F-OLLOWJNG—PROCEDURE-
        LEV«-LEV-BRIOOOOO
```

```
↓ 36↓
        MARKJUMP[ATLAS];                  | ASSIGN  EVERYTHING
        Q1 → DUMPWIDTH←LXPR2+Q1$;
        STORLOC > MAX → ACC ← STORLOC ; ACC ← MAX $ ;
        L18 ← ACC; CODE(STOP)
↓ 37↓
        STORLOC > MAX → MAX ← STORLOC $ ; | FIND  LAST  LOCATION  IN  FIXED S
        CLUTCH ← TRUE  ;                  | ONLY  NECESSARY IF PROCEDURE BODY
        MARKJUMP[ATLAS] ;                 | ASSIGN  LABELS, PROCS,ETC.
        LEV←LEV-8R100000;                 | RESTORE  LEVEL
        CXT → MARKJUMP[SASS] ; CODE(MARKJUMP[X86]);
              CODE(MARKJUMP[<X33>]) $;
        ENTER[SYMB;STAB];POP[STAB,0]; | ENTER  SCATTER  LABEL
        POP[STAB,STORLOC]             | RESET  FOR  OUTER  BLOCK
        ;  POP[STAB,LOC[LAB]]
↓ 38↓
        MARKJUMP[8R11765];
        JUMP[EXE31]
↓ 39↓
         FORV←RIGHT2;
        ALFA ← CODELOC;
        PUSH[FLAD3,0]
↓ 40↓
        T ← ABVAR;
        TYPE ← DOUBLE;
        VAL2 ← LEFT1;          | VAL2 HAS NOW THE POSTFIX INTEGER OF STEP
        MARKJUMP[DECLARE];
        PUSH[FLAD1,0]; PUSH[FLAD2,0];
        CODE(MARKJUMP[FLAD1]; JUMP[FLAD2]);
        ALFA←CODELOC; ASSIGN[FLAD1]; TALLY[CODELOC]
↓ 41↓
        CODE( JUMP[<ALFA>]);  ASSIGN[FLAD2]
↓ 42↓
        CODE( JUMP[<ALFA>]); T1 ← CODELOC;
        CODE( MARKJUMP[ALFA] );
        MARKJUMP[INCRE];
        ALFA←T1; ASSIGN[FLAD2]
↓ 43↓
        XEQ 112 = Q112 → LEV ← LEV;
        XEQ 112 ← Q112;
        CODE(MARKJUMP[<X122>]) $ ;
        PUSH[FLAD4,0];CODE( JUMP[FLAD4]);
        ASSIGN[FLAD3]; PUSH[FLAD4,CODELOC]; TALLY[CODELOC]
| 44↓   SEE DESIGNATIONAL EXPRESSIONS
↓ 45↓
        MARKJUMP[8R11763];
        ASSIGN[FLAD2]
| 50↓   SEE DESIGNATIONAL EXPRESSIONS
| 51↓   SEE DESIGNATIONAL EXPRESSIONS
| 52↓   SEE DESIGNATIONAL EXPRESSIONS
↓ 60↓
        RIGHT2 ← VAL2
↓ 61↓
        BETA ← CODELOC;
        PUSH [BASE,0];                | A  NEW  BASE
```

```
            LEFT2 ← FORV;
            MARKJUMP[FIND];
            ACC = 1 → JUMP[VARIABLE] $; FAULT 61
      ↓ 64↓
            RIGHT2 ← LEFT3;

            RIGHT1 ← LEFT2;
   SN       COR  0          5110063226
 14 SN      COR  0          0170062110
 15 SN      COR  0          4150063212
 16 SN      COR  0          0650000001
 17 SN      COR  0          1730063212
 1120       LEV←LEV
      | 65↓    SEE ARRAYS
      | 70|    SEE EVAL
      ↓ 71↓
            CODE(X1←0); CODELOC←CODELOC-2;  |DOMMY TO SAVE (IF NECESARY) THE ACC
            CODE( MARKJUMP[<X52>]);
            MARKJUMP[PUSEV]
      | 72↓    SEE EVAL
      ↓ 73↓
            'EXEC73'
            CODE( ACC ← LEFT2 );
            ACC←((LEFT2∧LAST)*8F1₈-7)-4;  | SHIFT  FOR TYPE NUMBER
   SN       COR             1330000000    | STZ  0   0;
51    SN    COR  0          1450000000
51.         TT ← ACC;
            <0>>0 → ACC←LXPR0:ACC←8L003257$;| LXP 0  R0  OR  LXM 0  R0
            ACC ← ACC ∨ TT;
            MARKJUMP[8R64341];
            ACC ← EVAL1 ∨ 8L041260;
            MARKJUMP[8R64341];
            ATTRIBUTE → CODE( MARKJUMP[<66>]) | CODE( MARKJUMP[<X53>]) $ ;
            ATTRIBUTE ← FALSE;  EVAL1 ← EVAL
      | 74↓    SEE EVAL
      ↓ 75↓
            → TEST[RIGHT1,SYMBOL] → FAULT 75 $
      | 76↓    SEE PATTERNS
      ↓ 77↓
            C←0;
            'EXE77'
            MARKJUMP[8R11743]
      ↓ 80↓
            C←1;
            JUMP[EXE77]
      ↓ 81↓
            RIGHT1 ← (TRU-LEFT1-1) ∨ <8R63306>
      | 82↓    SEE PATTERNS
      | 83↓    SEE PATTERNS
      | 84↓    SEE PATTERNS
      | 85↓    SEE PATTERNS
      | 86↓    SEE PATTERNS
      | 87↓    SEE PATTERNS
      | 91↓    SEE DESIGNATIONAL EXPRESSIONS
      ↓ 94↓
```

```
            T←SYMB[LEFT3,S,,];
            SIGNAL →
            ARRAY ⇒ (T^TMASK) →
                DOUBLE ⇒ (T ^¬TMASK) → ACC←0:ACC←1$; C←ACC;
            TT←SYMB[0,,S,];
            CODE(ACC←LEFT3);
            ACC ← LXPRO ∨ TT;
            MARKJUMP[8R64341];
            ACC ← 8L00126 ∨ C;
            MARKJUMP[8R64341];
            ACC ← 8L001261 ∨ (( T ^¬ TMASK)*8R1070000001);
            MARKJUMP[8R64341];
            CODE( MARKJUMP[<X59>]);
            MARKJUMP[ PUSEV ];
            CODE(STORLOC ← X3); TALLY[STORLOC] :
            FAULT 94 $ : FAULT 941 $
     ↓ 95↓
            RIGHT1 ← EVAL + 1 + MODE1 + FORMULA ;
            POP[EVAL,0]; EVAL1 ← EVAL
     ↓ 98↓
            ¬ TEST[LEFT2,FORMULA] → FAULT 98 :
            CODE( X3←LEFT2; X2←LEFT4); MINUS[CODELOC];
            CODE( MARKJUMP[<X36>]);
            MARKJUMP[8R11775];           | VALUE1←ACC,FORM
   SN       COR  0         1
L123 SN     COR  0         8
L17         LEV ← LEV $
     ↓ 99↓
            RELA ← BASE;
            MARKJUMP[8R11710]
     ↓100↓
            'EX100'
            MARKJUMP[UNMAKE2];
            MARKJUMP[UPSET];
            MARKJUMP[8R11660]
     ↓105↓
            MARKJUMP[UNMAKE2];
            MARKJUMP[SETTUP];
            MARKJUMP[8R11717]
     ↓107↓
            MARKJUMP[UNMAKE1];
            RELB ← BASE;
            TEST[LEFT2,DOUBLE]∨TEST[LEFT2,SINGLE]∨TEST[LEFT2,INTEGER] →
            C←0; CODE(X1←LEFT2); MINUS[CODELOC] :
            TEST[LEFT2,THUMP] →
            C←1; MARKJUMP[8R11723] :
            TEST[LEFT2,FORMULA] →
            C←1; MARKJUMP[8R11733] :
            FAULT 107 $ $ $
     ↓112↓
            RIGHT2   ←   LEFT2   ;        | RIGHT2  HAS  THE  WRONG  VALUE
     'STORE'
            CONST[LEFT4] → FAULT 712 :     | CAN'T STORE INTO A CONSTANT
            LEFT4 < 2000 → RUDY←FALSE;
            RELA ← BASE ; LEFT2 ← LEFT4 ; | STORE  MIGHT  USE  UPSET
```

```
        MARKJUMP[FIND] ;                  | GET  IT  OUT  OF  SYMB
        ACC = 1 →                          | VARIABLE  - DYNAMIC  OR  FIXED
  SVAR'      RELB←RELOC;TT←TYPE+TEMP+KEY:|STEREOTYPE  CONSIDERED  RELATIVE
        ACC = 5 →                          | FORMAL PARAMETER  CALLED  BY  NAME
            MARKJUMP[SAFEN];TT←TYPE+RZ;  | LOC[VARIABLE] WILL BE IN  RO
            CODSTK←TAR;TALLY[CODELOC];| TRM      V203
            CODSTK←(THUNK+KEY)∘SHIFT+RELOC;TALLY[CODELOC]:
        ACC = 6 → JUMP[SVAR]S;             | FUNCTION  NAME
        FAULT 112; JUMP[EXIT] S $ ;        | NOTHING  ELSE  WORKS
        LEFT2 ← RIGHT2; LEFT4←TT: RUDY←TRUE$;
        TEST[LEFT2,THUMP] →
        MARKJUMP[8R11712];
        CODE( MARKJUMP[<X54>]):
         TEST[LEFT4,SINGLE] ∨ TEST[LEFT4,LOGICAL] ∨
        TEST[LEFT4,DOUBLE]∨TEST[LEFT4,INTEGER] →
         TEST[LEFT2,SINGLE] ∨ TEST[LEFT2,LOGICAL] ∨
        TEST[LEFT2,DOUBLE]∨TEST[LEFT2,INTEGER] →
        CODE(LEFT4 ← LEFT2): FAULT 512$:
        TEST[LEFT4,BOOLEAN] →
        TEST[LEFT2, BOOLEAN] → CODE(LEFT4 ← LEFT2): FAULT 612$:
        TEST[LEFT4,FORMULA] →
        TEST[LEFT2,SYMBOL] →
        LEFT4←LEFT4-1;
        MARKJUMP[8R11753] :
        TEST[LEFT2,FORMULA] → CODE(LEFT4 ← LEFT2):
        MARKJUMP[8R11765];
        TEST[LEFT2,SINGLE] ∨
        TEST[LEFT2,DOUBLE]∨TEST[LEFT2,INTEGER] →
         CODE(MARKJUMP[<X21>]):
         TEST[LEFT2,LOGICAL]→CODE(MARKJUMP[<X24>]):
         TEST[LEFT2,BOOLEAN]→CODE(MARKJUMP[<X31>]):
         FAULT 112 $ $ $ ;
        CODE(LEFT4←X1); MARKJUMP[8R11655] $$:
        TEST[LEFT4,SYMBOL] →
        MARKJUMP[DT];
        CODE(MARKJUMP[<X101>]):
        FAULT 112 $ $ $ $ $ $
        ;    RUDY ← FALSE
 ↓113↓
        TEST[LEFT4, SYMBOL] →
        CODE(MARKJUMP[<X121>]):           |RECOVER PHANTOM
        BASE  ←  CXT  ;                   | IT  MIGHT  GO  INTO  A  TEMP
        CODE(VALUE2←ACC);RIGHT2←RIGHT2∨LEFT2∧LAST   | DON'T THROW VALUE AWAY
        $
 ↓114↓
        C=0 → CODE(VALUE2 ← LEFT4∨LEFT2); JUMP[SALIDA]:
        C←22; JUMP[FINAL] $
 ↓115↓
        C=0 → CODE(VALUE2 ← LEFT4∧LEFT2); JUMP[SALIDA]:
        C←21; JUMP[FINAL] $
 ↓116↓
        TEST[LEFT2,BOOLEAN] ∨
        TEST[LEFT2,LOGICAL] → CODE(VALUE2 ← ¬LEFT2); JUMP[SALIDA]:
        TEST[LEFT2,FORMULA] → CODE(X1←LEFT2);
        C←20; JUMP[FINAL] :
```

```
        FAULT 116 $ $
   17↓
        C=0 →
        CODE(VALUE2 ← LEFT4 <  LEFT2);
        C←15; JUMP[FINAL] $
↓118↓
        C=0 →
        CODE(VALUE2 ← LEFT4 >  LEFT2);
        C←14; JUMP[FINAL] $
↓119↓
        C=0 →
        CODE(VALUE2 ← ¬(LEFT4 < LEFT2));
        C←17; JUMP[FINAL] $
↓120↓
        C=0 →
        CODE(VALUE2 ← ¬(LEFT4 > LEFT2));
        C←16; JUMP[FINAL] $
↓121↓
        C=0 →
        CODE(VALUE2 ← LEFT4 ≠  LEFT2);
        C←19; JUMP[FINAL] $
↓122↓
        C=0 →
        CODE(VALUE2 ← LEFT4 =  LEFT2);
               C = 8R3 →
        CODE(MARKJUMP[<X186>]);
        VALUE2←ACC);
        SET[RIGHT2, BOOLEAN];
        C←18; JUMP[FINAL] $
        $
↓123↓
        C=0 → CODE(VALUE2 ← LEFT4+LEFT2); JUMP[SALIDA];
        C←12; JUMP[FINAL] $
↓124↓
        C=0 → CODE(VALUE2 ← LEFT4-LEFT2); JUMP[SALIDA];
        C←13; JUMP[FINAL] $
↓125↓
        C=0 → CODE(VALUE2 ← LEFT4*LEFT2); JUMP[SALIDA];
        C←10; JUMP[FINAL] $
↓126↓
        C=0 → CODE(VALUE2 ← LEFT4/LEFT2); JUMP[SALIDA];
        C←11; JUMP[FINAL] $
↓127↓
        C=0 → CODE(VALUE2 ← -LEFT2);
        C←32; JUMP[FINAL] $
↓128↓
        C=0 → CODE(VALUE2 ← LEFT4↑LEFT2); JUMP[SALIDA];
        C←09; JUMP[FINAL] $
↓131↓
        C=0 → CODE(MARKJUMP[<X60>]); JUMP[ACC2] ;
        C←06; JUMP[FINAL] $
↓132↓
        C=0 → CODE(MARKJUMP[<X61>]); JUMP[ACC2] ;
        C←05; JUMP[FINAL] $
↓133↓
```

```
        C-0 - C00EU1ARKJUMPI<X62>) )> JUMPlACC21 t
        C*04; JUMP(FINALJ $: . . . . . . . . . . . . . .
   .34*
        .C«0 .•»-C00E<MARKJUMPt<X63>) >l JUMPUCC2! I
        0 0 3 ; JUMPU'lNALJ S-
- ,135*-
        C = 0 * C00E<MARKJUMPt<X64>) > J JUMPUCC21 I
   _____C«-02J-JUMP(FiNAL)-S . . . . . . . . . . . . . .
   136*
   _____CO *~C00E<MARKJUMP(<X65>1 >J JUMPUCC21 . I.
        C-Oli JUMPIFINAU «•
   137*-
        C=0 - MAHKJ'Mpl8RH735U JUMPIACC2I »
._______OOOJ-.JUMP(FINALI--S ____________________
^1  140    SEE ARKAYS
_ |_.i.41-._,SEE-ARRAYS. . . . . . . . . . . . . . -
 i  142    SEE ARRAYS
__1_1 4 3_SEE-ARHAYS. . . . . . . . . . . . . . . . . . .
    144    SEE ARRAYS
 I  145_____SEE-ARRAYS________-. . . . . . . . . . . . .
  *146*
        ..T-YPE-        D0UBLE-
  *147*
        -T-YPE-.        -INTEQER-
^148*
        -TYPE.    >....BOOLEAN-
  *149*
        TYPE.     >....- LOGICAL
  *150*
        -T-Y-P-E-  _«•___FORMULA-
  *151*
        -TYPE-          .SYMBOL-
  *152*
        -TYPE_____>-   .SINGLE-
^*157*
        _FNO..>_2UL                          I-START COUNTING-PARAMETERS-
        XEQ 190      FLST                       FORMAL PARAMETER LIST
-*159*
        PUSH[STA3,STORL0Cj;ST0RL0C.-1J  | RESET STORAGE BEFORE SEEING FUMCTI
        -« CLUTCH*- PUSH(FLAD4.0J;CODE(JUMPIFLAD4) >).....- ________________
             CLUTCH * TRUE S            ! JUMP AROUND PROCEDURES
_p*160*-
        RIGHT2-CXT'R1GHT3-ACC;          | R3 FOR FUNC.J R2 FOR PROC
     -.....CXT. <•....CODELOC   ;. . . . .I NOW WE HAVE-THE-NEW CONTEXT.......
        COR          0737000000   STZ      O'CODELOCJ
     -__________________-.....TALLYICODELOC;t \ ONE W RD FORTHE....CONTEXT..........
        PUSH{LSS,CSSJ;CSS*CODELOC;      | SET UP AN ORIGINAL HEAD OF CHAIN
     _-<CODELOC>.«-LEy;JALL-YtCODELOC) /- I KEEP LEVEL IN THE...HEAD___:__________
        LEV *> LEV   BR100000 ;         I KEEP LEVEL IN THE HEAD
     ___T -......FUNCTION....t . . . . . . . . . .I SHOWING THAT IT ISN'T_________~_......
        RIGHT1   LEFT1 ;                \ SAVE "HE IDENTIFIER
     ___SETtLEFT1,FUNCTION];            | THE TAGGED IDENTIFIER......_.....
        PUSHtLADLE.LbFT1J;              I INTO THE POT FOR ATLAS
     _____PUSHJLAOLE.CXT].....;. . . . . . . .I IN THIS CASE A PROCEDURE-NAME-....
         PUSH[LADLE* U  ]               I FUNC. DESIG. GLOBAL TO FUNC,
```

```
 +161+
        F+STORLOC;TALLY(STORLOC);        | NORMAL  RESERVATION
        TYPE=DOUBLE+TALLY(STORLOC)S;     | TWO  WORDS  REALLY
        T  +  TYPE  +  PRCDR             | IT'S  SOME  KIND  OF  FUNCTION
 +162+
        ENTER(SYMB;RIGHT1,T,F,CXT);      | FINALLY  THE  ENTRY
        PUSH(STAB,8L2+LOC(SYMB));        | REMEMBER  THE  SCATTER  LABEL
        RIGHT1 + CODELOC                 | IS  THIS  USEFUL  NQQQQQQQQQQQQQO
CO +163+ IN  SYNTAX  BUT  NOT  NEEDED  NOW
CO +164+ IN  SYNTAX  BUT  NOT  NEEDED  NOW
 +165+
        CLUTCH + ASSIGN(FLAD4);CLUTCH+FALSE$ | BACK  TO  THE  STATEMENTS
 +167+
        XEQ 190 + SLST                   | SPECIFIER  LIST
 +172+
        XEQ 190 + VLST                   | VALUE  LIST
 +174+
        XEQ 190 + ENEX                   | VARIABLE  LIST

                      | ROUTINES TO PROCESS ARRAYS

 + 16+
        C+1;
        'SIGUE'
        T+SYMB(RIGHT2,$,,);
        SIGNAL +
        ARRAY =(T ^ TMASK) +
      TT + SYMB(0,,$,);
        RIGHT2 +(T ^~ TMASK);
      ACC = DOUBLE + C+C+2 $;
      CODE(ACC+<TT>);
      ACC + 8L001263 v C;
        MARKJUMP(8R64341);          | LOAD:  LXP 0  (1,2,3,0),/63
        CODE(MARKJUMP(<X43>)); FAULT 16$;FAULT 7$
 + 17+
        MARKJUMP(8R11704)
 + 25+
        MARKJUMP(8R11765); CODE(MARKJUMP(<X44>))
 + 65+
        C+0; JUMP(SIGUE)
 +140+
        PUSH(FLAD2,0); CODE(JUMP(FLAD2));
        ASSIGN(FLAD1); ALFA+CODELOC;
        TALLY(CODELOC); CODE(MARKJUMP(<X40>))
 +141+
        CODE(Y1+LEFT2; Y2+LEFT4); MINUS(CODELOC);
        CODE(MARKJUMP(<X41>))
 +142+
        'ENTRA'
        PUSH(FLAD1,0);BETA+TYPE;
        T + ARRAY;
        RIGHT1 + TYPE;
        XEQ 190 + ALST
 +143+
```

```
        TYPE ← DOUBLE; JUMP[ENTRA]
 44↓
        TYPE ← LEFT2;
        JUMP[ENTRA]
↓145↓
        BETA = DOUBLE → CODE(ACC←1) ↑ CODE(ACC←0) $ ;
        CODE(MARKJUMP[<X42>]; JUMP[<ALFA>]);
        ASSIGN[FLAD2]
                    |    ROUTINES TO PROCESS EVAL

↓ 70↓
        TEST[RIGHT2,FORMULA] ∨ TEST[RIGHT2,SYMBOL] →
        CODE(X1←RIGHT2); MINUS[CODELOC];
        CODE( MARKJUMP[<X51>]);
        MARKJUMP[8R11775];          |   VALUE2 ← ACC, FORM
N       COR  0            2
N       COR  0            8
        RIGHT2 ← (RIGHT2 ∧ <8R11737>) ∨<8R63304>:
        FAULT 70 $
↓ 74↓
        RIGHT2 ← EVAL + MODE1 + SYMBOL ;
        POP[EVAL,EVAL1];
        JUMP[EXEC73]
↓ 72↓
        TEST[LEFT5,SYMBOL] ∧ TEST[LEFT2,SYMBOL] →
        TEST[LEFT4,FORMULA] →
        CODE(Y4←LEFT2; Y3←LEFT5; X1←LEFT4); MINUS[CODELOC];
        CODE(MARKJUMP[<X38>]);
        MARKJUMP[8R11775];          |   VALUE1 ← ACC, FORM
N       COR  0            1
N       COR  0            8
        RIGHT1 ← (RIGHT1∧<8R11737>)∨<8R63304>:
        RIGHT1←LEFT4$: FAULT 72$

       |    ROUTINES FOR DESIGNATIONAL EXPRESSIONS

↓ 91↓
        T←LAB[LEFT2,,,,$];
        SIGNAL →
        T=0 → FAULT 91:
        LAB[0,,,,$] ← 0;
        ASSIGN[LOC[LAB[0,,$,,]]] $ ;
        T←CODELOC;
        ENTER[LAB;LEFT2,LABL,T,LEV,0] $
→ 44↓
        'PRINCI'
        T ← LAB[LEFT2,,,,$];
        SIGNAL →
        LAB[0,$,,,] = LABL →
        COMT 2 ← LOC[LAB[0,,$,,]];
        T = 0 →
```

```
        COMT 3 ← <COMT 2>;
        CODE(JUMP[COMT 3]) :
        CODE(JUMP[CHAIN[COMT 2]]) $ : FAULT 44 $ :
        ENTER[LAB; LEFT2, LABL, 0, LEV, 1];
        JUMP[PRINCI] $
↓ 50↓
        ENTER[LAB; LEFT2,SWIT,STORLOC,LEV,0];
        BETA←STORLOC;
        SWCONT←1; TALLY[STORLOC];
        'EX50'
        PUSH[FLAD4,0]; T← CODELOC+3;
        CODE(STORLOC ← LOC[T]; JUMP[FLAD4]);
        TALLY[STORLOC]
↓ 51↓
        SWCONT ← SWCONT+1;
        ASSIGN[FLAD4]; JUMP[EX50]
↓ 52↓
        ASSIGN[FLAD4]; CODE(BETA ← LOC[SWCONT])
↓ 15↓
        TYPE ← LAB[LEFT4,$,,,];
        SIGNAL →
        TYPE = SWIT →
        T←LAB[0,,$,,];
        CODE(Y1←LEFT2; Y2← T); MINUS[CODELOC];
        CODE( JUMP[<X35>]) : FAULT 15 $ : FAULT 7 $

    |   ROUTINES FOR PATTERNS

↓ 76↓
        MARKJUMP[8R11751]
↓ 82↓
        TYPE ← FUNCTION
↓ 83↓
        MARKJUMP[8R11715]
↓ 84↓
        CODE( VALUE1←LEFT2 ∨ <X58>);
N       COR   0            0326000001
N       COR   0            6156000070
N       COR   0            5350063245
N       COR   0            3736000070
N       COR   0            4150063342
N       COR   0            5350063301
N       COR   0            1730063342
        SET[RIGHT1,FORMULA]
↓ 85↓
        TEST[LEFT4, SYMBOL] →
        ¬ TEST[LEFT2, SYMBOL] →
        MARKJUMP[DATATERM];
        CODE(MARKJUMP[<X136>]) $ ;
        JUMP[INST] $ ;
        C←0;
        'EXE85'
        MARKJUMP[8R11745]
```

```
 + 86+
        C+1;
        JUMP[EXE85]
 . 87+
        MARKJUMP[8R11747]
  190+ FAULT  190  ;                    | USED  ONLY  AS  A  CARRIER

          |-*-*   RUDY'S ROUTINES   *-*-|

 'DECLARE''
        LH  + STORLOC;                  |   SAVE LOC
        TALLY[STORLOC];                 | NORMAL  ALLOCATION
        RH+STORLOC ;                    | SAVE  LOC[SECOND HALF OF WORD]
        T≠ARRAY →
        ACC + TYPE;
        ACC = DOUBLE → TALLY[STORLOC] ;
        ACC = FORMULA → TALLY[STORLOC];
        LEFT1 + LEFT1 - C1;
        CODE[ACC+LEFT1];
        MARKJUMP[<X34>];
        LH+X1]; MARKJUMP[V60]]
        LEFT1 + LEFT1 + C1;
        CODE[RH+X2];
        ACC=SYMBOL →
        CODE[X1+LEFT1;
        MARKJUMP[<X105>]; LH+X1] $ $ $ $ ;
        ENTER[SYMB; LEFT1,TYPE+T,LH,CXT]];  |  ID,TYPE,KEY,RELOC
        JUMP[<DECLARE>];
 ''FIND''
        T+SYMB[LEFT2,S,,];TYPE+ACC^¬TMASK; | FIND ENTRY AND GET TYPE
        ¬SIGNAL→FAULT 191;NFALTS+NFALTS-1;| NOT  REALLY  AN  ERROR
            ACC + -1; JUMP[<FIND>] $  ;  | GO  BACK  SAYING  SO
        KEY + SYMB[0,,$,] ;             | RELATIVE  ADDRESS
        RELOC+SYMB[0,,,S] ;             | RELOCATION  BASE
        ACC + T / 64 ^ 7 ;              | GET  CODE  DIGIT
        JUMP[<FIND>] ;                  | -30-
 'ATLAS''        | ASSIGNS  LABELS,PROCEDURES,ETC.
    'NEWN' POP[LADLE,TT];              | < T , TT > == < ID' , VALUE >
        TT≠0→POP[LADLE,T ];            | UNLESS  IT  IS  A DELIMITER
            TEST[T,LABEL    ]→TTT+RAG;ACC+LABLE:| LABEL
            TEST[T,FUNCTION]→TTT+PAR;ACC+PRCDR:| PROCEDURE
            TEST[T,THOUGHT ]→TTT+TAR;ACC+THUNK:| PARAMETER--LABEL,PROCEDURE
            FAULT 391 S S S ; IV + ACC * SHIFT ; |
            T   +   X7   ^   T   ;   | CLEAR  T
            A+LOC[CHADLE[T,S]];C+<A>^X7;  | INITIALIZE ASSIGNMENT LOOP
            SIGNAL  →                  | DON'T  WORRY  IF  IT  WASN'T  USED
 ASGN'          CFCHEND→B+C;C+<B>^X7;      | GET  NEXT  ELEMENT
                    <B><LEV → A + B ; | MOVE ALONG CHAIN IF ILLEGAL
                    <A>+<A>^¬X7+C;      | DELETE  <B>  FROM  CHAIN
                    (<B>^MODE1)≠0→<B>+TT∨IV;| CHAINING  OF  PARAMETERS
                    <B>+TT;B+B-1;<B>+TTT $$;| ASSIGNING  OTHER  CALLS
                JUMP[ASGN] $  $  ;  | CHECK  ANOTHER  ELEMENT
```

```
          JUMP[NEWN] $ ;            | TRY  ANOTHER  CHAIN
     JUMP[<ATLAS>] ;                | GO BACK
''SASS''   | SIZE  AND  SHAPE  SETTER  OF  BLOCK  LENGTH
     T←<CSS>∧X7;<CSS>←(<CSS>∧¬X7)+STORLOC; | INSERT  THE  LENGTH  IN  THE  C
'SAS' T→TT←<T>*R15;<T>←(<T>∧X7)-STORLOC;| ASSIGN  SIZE  OF  INNER  BLOCK
          T ← TT ; JUMP[SAS] $ ;    | GO  DO  ANOTHER  ONE
     POP(LSS,CSS);JUMP[<SASS>]    ; | POP  AND  LEAVE
''CALL''            | COMPILES  A  CALL  ON  A  PROCEDURE
     CODSTK ← ERROR ; TALLY(CODELOC);| ERROR  IF  UNASSIGNED
     MARKJUMP[HEAD];                | FIND OR CREATE  HEAD  OF  A  CHAIN
     CODSTK←LEV∨CHAIN[<T>];TALLY(CODELOC); | PAR'S  PARAMETER
     JUMP[<CALL>] ;
''HEAD''            | FINDS OR CREATES IN T THE ENTRY FOR LEFT2
     T ← LOC[CRADLE[LEFT2,S]] ;     | GET  CHAINING  ADDRESS
     ¬SIGNAL→ENTER[CRADLE;LEFT2,CHEND];| PUT  'ER  THERE
          T ← LOC[CRADLE]-320 $ ; | GET  THE  CORRECT  CHAINING  ADDRESS
     JUMP[<HEAD>] ;
''SETTUP''  POP[BASE,RELB];RELA←BASE; | SET  UP  THE  BASES
     BASE ← CXT ; JUMP[<SETTUP>];   | POSSIBLY  TEMP  STORE
''UPSET''  POP[BASE,RELA];RELB←BASE; | SET UP  REVERSELY
     BASE ← CXT ; JUMP[<UPSET>];    | AGAIN  FOR  TEMPS
''INCRE''                     | COMPILE;   FORV ← FORV + INCRE
     LEFT2 ← FORV;
     MARKJUMP[FIND];
     ACC = 1 → TT ← KEY + MODE1 + TYPE + TEMP;
     RELB ← RELOC;
     RELA ← CXT;
     LEFT2 ← VAL2;
     MARKJUMP[FIND];
     T ← KEY + MODE1 + TYPE + TEMP;
     CODE(  TT ← TT + T );
     JUMP[<INCRE>] ; FAULT 999 $ ; JUMP[EXIT] ;
''DATATERM''
     CODE(ACC←LEFT2);
     TEST[LEFT2, BOOLEAN] → CODE(MARKJUMP[<X31>]);
     TEST[LEFT2, INTEGER] ∨ TEST[LEFT2, SINGLE]
     ∨ TEST[LEFT2, DOUBLE] → CODE(MARKJUMP[<X21>]);
     TEST[LEFT2, LOGICAL] → CODE(MARKJUMP[<X24>]);
     TEST[LEFT2, FORMULA] → CODE(X1←X2);
     MARKJUMP[8R11655] $ $ $ $ ;
     CODE(MARKJUMP[<X151>]);
     JUMP[<DATATERM>];
''UNMAKE1''
     TEST[LEFT2, SYMBOL] →
     CODE(MARKJUMP[<X205>]);
     VALUE2 ← ACC);
     LEFT2 ← RIGHT2;
     SET[LEFT2, TRUMP] $ ;
     JUMP[<UNMAKE1>];
''UNMAKE2''
     MARKJUMP[UNMAKE1];
     TEST[LEFT4, SYMBOL] →
     CODE(MARKJUMP[<X205>]);
     VALUE2 ← ACC);
     LEFT4 ← RIGHT2;
```

```
      SET[LEFT4, TRUMP] $ ;
      JUMP[<UNMAKE2>];
   T''
      ← TEST[LEFT2, SYMBOL] →
      MARKJUMP[DATATERM];
      CODE(MARKJUMP[<X136>]) $ ;
      JUMP[<DT>];
      ''PUSEV''
      CODE( STORLOC ← X1 )]  MARKJUMP[ V60 ]]
      EVAL1 ← STORLOC;
      PUSH[EVAL,EVAL1];
      TALLY[STORLOC];
      JUMP[<PUSEV>];
      'SALIDA'
      RIGHT2 ← RIGHT2 ∧ <8R11737>;
      TEST[LEFT2,LOGICAL]∨TEST[LEFT4,LOGICAL]→SET[RIGHT2,LOGICAL]:
      TEST[LEFT2,DOUBLE ]∨TEST[LEFT4,DOUBLE ]→SET[RIGHT2,DOUBLE ]:
      TEST[LEFT2,SINGLE ]∨TEST[LEFT4,SINGLE ]→SET[RIGHT2,SINGLE ]:
      TEST[LEFT2,BOOLEAN] → SET[RIGHT2,BOOLEAN] :
      SET[RIGHT2,INTEGER] $ $ $ $ ;
      JUMP[EXIT];
      'ACC2'
      MARKJUMP[8R11775];      |  VALUE2 ← ACC, REAL
      COR  0            2
      COR  0            3
      JUMP[EXIT];
      'FINAL'
     MARKJUMP[8R11652]
      ;
FAULT 990  ← END
```

REFERENCES

Feldman, J. A., "A Formal Semantics for Computer Languages", doctoral dissertation, Carnegie Institute of Technology. (1964).

Feldman, J. A., "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler", Communication of the ACM, Vol. 9, p 3 (Jan. 1966).

Perlis, A. J. and Iturriaga, R., "An Extension to Algol for Manipulating Formulae", Communications of the ACM, Vol. 7, p 127 (Feb. 1964).

Perlis, A. J., Iturriaga, R., and Standish, T., "A Definition of Formula Algol", Computation Center, Carnegie Institute of Technology, (March 1966).

Iturriaga, R., Standish, T., Krutar, R., and Earley, J., "Techniques and Advantages of Using the Formal Compiler Writing System FSL to Implement a Formula Algol Compiler", Proc. SJCC, p 241 (May 1966).