This document was produced by SDC and III in performance of contract AF 19(628)-5166 with the Electronic Systems Division, Air Force Systems Command, in performance of ARPA Order 773 for the Advanced Research Projects Agency Information Processing Techniques Office & Subcontract 65-107.

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 200 Sixth Street / Cambridge, Massachusetts 02142

TM-2710/331/00

AUTHOR Erwin Book

TECHNICAL C. Weissman

RELEASE S. L. Kameny

for J. I. Schwartz,

DATE 15 April 1966

PAGE 1 OF 27 PAGES

The LISP 2 Syntax Translator

ABSTRACT

This document presents a description of the syntax of the LISP 2 source language (SL) and describes its translation into LISP 2 intermediate language (IL). The interpretation of the source language in terms of intermediate language is referred to as the semantics of the source language. SDC document, TM-2710/220/00, dated 4 November 1965, describes the semantics of the LISP 2 intermediate language.

1. INTRODUCTION

The language presented herein is described through the eyes of the syntax translator. The LISP 2 syntax translator translates SL to IL. The syntax translator is produced by feeding the syntax equations describing the syntax and semantics through the LISP version of the meta compiler. Briefly, the meta compiler accepts as input a BNF-like source language and translates this language to a list structure. As a matter of fact, this list structure is a LISP 1.5 program which is a compiler for the language being defined. These compilers are models of an extended push-down store automaton and use one main stack which is the pseudo accumulator of this machine. The accumulator is called the star stack and is symbolized by *. At any time the star stack may hold a number of constructs. These may be referred to by name: to wit, *1, the top construct; *2, the second construct. A more complete description of the meta compiler is presented in SDC document TM-2710/330/00, dated 2 November 1965.

2. DEVELOPMENT CYCLE

The development cycle of the translator is as follows:

In Phase 0 of the project, the meta compiler is developed as a LISP 1.5 program. When presented with translation specifications in the form of syntax equations, the LISP 1.5 version of Meta produces compilers as LISP 1.5 programs.

During Phase 1, the syntax translator for LISP 2 is written and run through the meta compiler. The resultant LISP 1.5 program (from the syntax equations) is checked out; these equations are described in this document.

During Phase 2, the meta compiler is modified in two ways. The first modification causes it to produce a LISP 2 intermediate language program instead of a LISP 1.5 language program. The second modification makes a particular program produced, called the LISP 2 syntax translator, interface with a program called the Finite State Machine (FSM), and is done to speed up translation. The FSM reads and recognizes only LISP 2 tokens. However, the use of the FSM restricts the generality of the meta compiler since it results in a compiler which only understands a fixed alphabet, to wit, LISP 2 tokens.

The official version of the meta compiler released for general use will not use the FSM to read the input tape. Instead, a language and processor for generating FSM programs can be produced.

In Phase 3, the LISP 2 version of the syntax translator is checked out.

Currently, Phase 2 is underway.

3. SYNTAX EQUATIONS

The remainder of this document describes the syntax translator for LISP 2. The complete set of translation specifications is exhibited in Appendix A. Numbers appearing to the left of each line are not part of the syntax equations but merely line numbers used by an editing program. The format followed in explaining the correspondence between LISP 2 source language and intermediate language is as follows:

- A title
- A syntax equation or equations
- Some examples of source language and the corresponding intermediate language
- Further explanations when new forms appear in the meta language

Almost all of the examples were verified by running the LISP 1.5 version of the LISP 2 syntax translator on the Q-32 computer. The equations on lines numbered higher than 163, CONSTANT, are not illustrated because they describe the basic tokens of LISP 2 (see SDC document TM-2710/210/00, LISP 2 Token Syntax) and will be read by the Finite State Machine program in the final LISP 2 version.

The equations are illustrated starting with the simplest and proceeding toward the more complex.

3.1 VARIABLE EQUATION

```
0016000-VARIABLE = IDENTIFIER ('$' (IDENTIFIER .[EXTERNAL,*2,*1]/
0016100-      '$' .[EXTERNAL,*1]) /.EMPTY );
```

Source	Intermediate	Comments
AB$YZ	(EXTERNAL AB YZ)	Alternative 1.1
X$$	(EXTERNAL X)	Alternative 1.2
EXTRA.LONG.NAME	EXTRA.LONG.NAME	Alternative 2.

This syntax equation shows that a variable is an identifier followed by either a $ or empty (referred to as Alternative 1 and Alternative 2, respectively). Alternative 1 consists of two cases; Alternative 1.1 and Alternative 1.2. Alternative 1.1 is that an identifier follows the $. Alternative 1.2 is that another $ follows the first $. Alternative 1.1 indicates that two identifiers have been recognized and pushed into the star stack (*). A list of three elements is produced: the first element is the word EXTERNAL; the

second element is the second construct in the star stack, namely the first identifier recognized; and the third element produced is the top construct, namely the second identifier recognized. In Alternative 1.2, only one identifier is in the star stack and a list of two elements is produced; the word EXTERNAL and the top construct in *. In Alternative 2 the identifier, which is recognized and put into * by the IDENTIFIER syntax equation, remains untouched. In the examples above, the three cases are illustrated in order.

3.2 NAME EXPRESSION EQUATION

```
0015900-NAME-EXP = VARIABLE ('(' CALL-PART ')' .[*2,-[*1]-]/.EMPTY);
0016200-CALL-PART = < F-EXP $ (',' F-EXP)\.EMPTY >;
```

Source	Intermediate	Comments
ABLE(X,Y,Z)	(ABLE X Y Z)	Alternative 1
T$$()	((EXTERNAL T))	Alternative 1
BAKER	BAKER	Alternative 2

A name expression is either a variable followed by a parenthesized call part, referred to as Alternative 1, or empty, referred to as Alternative 2. The definition of a call part is enclosed in angle brackets, < >. These brackets are used to collect as a list, in a first-in first-out manner, all constructs which are described by the equation within the brackets. That is, an F-EXP is followed by a sequence which may be empty of ',' F-EXP's. A call part may be empty. In Alternative 1, after the ')' is recognized, there are only two constructs in *. The second construct is the variable; the top construct is a list of an indefinite number of objects. The example given above shows a list of two elements being constructed. The first is the variable. The second -[*1]- means that the individual elements of the top construct of * should be inserted, thus forming a list of indefinite length whose first element is the variable recognized. The examples illustrate the cases. Note that the second example contains an empty call part.

An example of the input tape and star stack during processing illustrates this process.

Input Tape	* Stack	Comments
ABLE(X,Y,Z) ↑	* = (ABLE)	
ABLE(X,Y,Z) ↑	* = (( ) ABLE)	The < in CALL PART causes this.
ABLE(X,Y,Z) ↑	* = ((X) ABLE)	
ABLE(X,Y,Z) ↑	* = ((Y X) ABLE)	
ABLE(X,Y,Z) ↑	* = ((Z Y X) ABLE)	
ABLE(X,Y,Z) ↑	* = ((X Y Z) ABLE)	The > in CALL PART causes this.

The execution of .[*2,-[*1]-] follows:

```
.[*2,-[*1]-]                          * = ((X Y Z))
   ↑
      ABLE
.[*2,-[*1]-]                          * = ( )
      ↑
      ABLE X Y Z
.[*2,-[*1]-]                          * = ((ABLE X Y Z))
         ↑
```

3.3 FULL LOCATIVE EQUATION

```
0015700-FULL-LOCATIVE = NAME-EXP ('←←' FULL-LOCATIVE .[LOCSET,*2,*1]/
0015800-      .EMPTY );
```

Source	Intermediate
X← ←Y← ← A(Y,Z) ;	(LOCSET X (LOCSET Y (A Y Z)))

This syntax equation, which describes a full locative, is an example of right recursion. This equation enters itself as long as there are '← ←' and produces lists of three elements by grouping from right-to-left. The example presented above illustrates this.

3.4 WORD LOCATIVE EQUATION

```
0015300-WORD-LOCATIVE = ↑'BIT' '(' EXPRESSION ',' EXPRESSION
0015400-      ',' WORD-LOCATIVE ')' .[*4,*3,*2,*1] /
0015500-      ↑'CORE' '(' EXPRESSION ')' .[*2,*1] /
0015600-      FULL-LOCATIVE;
```

Source	Intermediate
BIT(X,Y,CORE(Z))	(BIT X Y (CORE Z))
CORE(BIT(A,B,C))	(CORE (BIT A B C))

The word locative has three alternatives, BIT, CORE and FULL locative. The "up" arrow in front of the string 'BIT' causes the following to happen. If the characters B, I, and T are next on the input tape, the atom BIT is pushed into * and the characters are bypassed; otherwise, both the input tape and * remain unchanged.

3.5 CR-EXPRESSION EQUATION

```
0015100-CR-EXP.. +'C' (+'A'/+'D') $(+'A'/+'D') +'R'
0015200-      MAKEATOM[];
```

Source	Intermediate
CAR	CAR
CADDADDAAR	CADDADDAAR

This is the first example of a syntax equation starting with .. instead of an =. This is used to form a token out of characters in a first-in first-out, manner. The significance of the + sign in front of a string is that if the input tape matches the string, the characters from the input tape are put into * as character atoms. The equation says a CR expression is a C followed by a A or D followed by zero or more A's or D's followed by an R. If such a combination exists, it ends up as a list of character atoms. MAKEATOM makes an atom of this list. The example shows that the characters making up the token have been collected in a first-in, first-out order.

An example of the input tape and the star stack during execution of this syntax equation follows:

Input Tape	* Stack
CAR ↑	* = (( ))

The .. after CR-EXP causes leading blanks to be deleted. Also, blip (i.e. ( )) is pushed into * to collect the token in a first-in first-out manner.

CAR ↑	* = (('C))
CAR ↑	* = (('C 'A))
CAR ↑	* = (('C 'A 'R))

Then MAKEATOM compresses the list of character atoms into an atom.

* = (CAR)

3.6 LIST LOCATIVE EQUATION

```
0014800-LOCATIVE = LIST-LOCATIVE\WORD-LOCATIVE;
0014900-LIST-LOCATIVE = (CR-EXP\↑'PROP') (LOCATIVE\'(' EXPRESSION ')'
0015000-      ) .[*2,*1];
```

Source	Intermediate
CADDDR FN(Y)	(CADDDR (FN Y))
PROP CAR X	(PROP (CAR X))

3.7 BASIC UNITS OF SOURCE LANGUAGE EQUATION

```
0014600-UNIT = NEGATIVE/CONSTANT/'(' EXPRESSION ')' /BLOCK/
0014700-      LOCATIVE ('←' F-EXP .[SET,*2,*1]/.EMPTY);
```

Source	Intermediate
NULL X	(NULL X)
3.16	3.16
3E2	3E2
77Q2	77Q2
TRUE	TRUE
(17)	(17)
BEGIN Y END	(BLOCK NIL Y)
X(I,J) ← CAR Y	(SET (X I J)(CAR Y))
MATRIX(U,V)	(MATRIX U V)

3.8 PRIMARY EQUATION

```
0014300-PRIMARY = '+' PRIMARY/'-' PRIMARY .[MINUS,*1] /
0014400-      ↑'ATOM' PRIMARY .[*2,*1] /
0014500-      UNIT ('↑' PRIMARY .[EXPT,*2,*1] /.EMPTY);
```

Source	Intermediate
+3.14	3.14
-X	(MINUS X)
ATOM L	(ATOM L)
A ↑B ↑2	(EXPT A (EXPT B 2))

3.9 FACTOR1 EQUATION

```
0014100-FACTOR1 = PRIMARY $(('/' ↑QUOTIENT PRIMARY\'\' ↑REMAINDER
0014200-      PRIMARY\'-:' ↑IQUOTIENT PRIMARY) .[*2,*2,*1] );
```

Source	Intermediate
W-:X/Y\Z$B	(REMAINDER (QUOTIENT (IQUOTIENT W X) Y) (EXTERNAL Z B))

A factor1 is left recursive; however, the meta language expression of this uses the sequence operator ($) to express this. The $ is read zero or more of the following expressions. The example shows that these constructs are grouped from left-to-right in pairs.

Factor1 is defined as a primary followed by a sequence which may be empty consisting of a:

- '/' substitute QUOTIENT followed by a primary
- '\' substitute REMAINDER followed by a primary
- '-:' substitute IQUOTIENT followed by a primary

Each time a pair of these primaries are recognized, a list consisting of the operator (*2), the first operand (*2) and the second operand (*1), is output. The first time *2 is performed, the second element of * was deleted, thereby converting the third element of * into the second element.

An example of the input tape and the star stack clarifies this.

Input Tape	* Stack
A/B ↑	* = (A)
A/B ↑	* = (QUOTIENT A)
A/B ↑	* = (B QUOTIENT A)

The .[*2,*2,*1] is executed as follows:

```
.[*2,*2,*1]                 * = ( B A )
  ↑
QUOTIENT
.[*2,*2,*1]                 * = ( B )
     ↑
QUOTIENT A
.[*2,*2,*1]                 * = ( )
        ↑
QUOTIENT A B
.[*2,*2,*1]                 * = ((QUOTIENT A B))
          ↑
```

3.10 FACTOR EQUATION

```
0013900-FACTOR = FACTOR1 ('*' < FACTOR1 $ ('*' FACTOR1) >
0014000-      .[TIMES,*2,-[*1]-] /.EMPTY);
```

Source	Intermediate
SIN(X) * 5.2\Z *77Q3	((TIMES (SIN X) (REMAINDER 5.2 Z) 77Q3))

This syntax equation shows that FACTOR's are grouped from left-to-right in pairs. This grouping is done to leave the decision as to the order of computation for multiplication in the intermediate language to the machine language translator, to wit, the compiler. The syntax equation shows that once the first '*' is detected, the rest of the terms in the series are collected and inserted in the output list individually, making a list of indefinite length.

3.11 SUM1 EQUATION

```
0013800-SUM1 = FACTOR $ ('-' FACTOR .[DIFFERENCE,*2,*1] );
```

Source	Intermediate
A-B$C-D$$	(DIFFERENCE (DIFFERENCE A (EXTERNAL B C)) (EXTERNAL D))

3.12 SUM EQUATION

```
0013600-SUM = SUM1 ('+' < SUM1 $ ('+' SUM1) >
0013700-      .[PLUS,*2,-[*1]-] /.EMPTY);
```

Source	Intermediate
G$B+H+A/B +K	(PLUS (EXTERNAL G B) H (QUOTIENT A B) K)

3.13 RELATION EQUATION

```
0013400-RELATION = SUM $(('/=' ↑NQ/'=' ↑EQ/'<=' ↑LQ/'<' ↑LS/
0013500-      '>=' ↑GQ/'>' ↑GR) SUM .[*2,*2,*1] );
```

Source	Intermediate
SIN(X) < .5	(LS (SIN X) .5)
CDR Y /= CAR X	(NQ (CDR Y) (CAR X))
A ↑2 >= B↑2	(GQ (EXPT A 2) (EXPT B 2))

3.14 NEGATION/NEGATIVE EQUATION

```
0013200-NEGATION = NEGATIVE/RELATION;
0013300-NEGATIVE = (↑'NOT'/↑'NULL') NEGATION .[*2,*1];
```

Source	Intermediate
NOT A	(NOT A)

3.15 INTERSECTION EQUATION

```
0013000-INTERSECTION = NEGATION ('AND' <NEGATION $ ('AND' NEGATION)>
0013100-      .[AND,*2,-[*1]-]/.EMPTY );
```

Source	Intermediate
NOT A < 2 AND B = SIN (Y)	(AND (NOT (LS A 2)) (EQ B (SIN Y)))

3.16 UNION EQUATION

```
0012800-UNION = INTERSECTION ('OR' <INTERSECTION $ ('OR' INTERSECTION)>
0012900-      .[OR,*2,-[*1]-]/.EMPTY );
```

Source	Intermediate
X OR Y > 2 OR Z AND S	(OR X (GR Y 2) (AND Z S))

3.17 SIMPLE EXPRESSION EQUATION

```
0012600-SIMPLE-EXPRESSION = UNION ('. ' SIMPLE-EXPRESSION
0012700-      .[CONS,*2,*1] /.EMPTY );
```

Source	Intermediate
X(A) . CAAADR B	(CONS (X A) (CAAADR B))

3.18 FUNCTIONAL EQUATION

```
0011800-FUNCTIONAL = 'FUNCTIONAL' '(' EXPRESSION ';' FUNARG-LIST ')'
0011900-      .[FUNCTIONAL,*2,-[*1]-] /
0012000-      (VALUE-TYPE 'FUNCTION' (VARIABLE\FALSIE) .[*1,*1]/
0012100-      'FUNCTION' (VARIABLE\FALSIE)) PAR-NAME-LIST (';'/.EMPTY)
0012200-      PAR-DECL-LIST .[-[*2]-,-[*1]-] PROCESS[] '(' EXPRESSION
0012300-      (';' FUNARG-LIST .[FUNCTION,*4,*3,*2,-[*1]-] /
0012400-      .EMPTY .[FUNCTION,*3,*2,*1] ) ')';
0012500-FUNARG-LIST = <VARIABLE $ (',' VARIABLE)\.EMPTY>;
```

Source

FUNCTIONAL (X+Y ; X,Y)

Intermediate

(FUNCTIONAL (PLUS X Y) X Y)

Source

REAL FUNCTION BLAH (A) INTEGER A; (X-Y*A;X,A)

Intermediate

(FUNCTION (BLAH REAL) ((A INTEGER))

(DIFFERENCE X (TIMES Y A)) X A)

3.19 CONDITIONAL EXPRESSION EQUATION

```
0011300
COND-EXPRESSION = 'IF' X-CLAUSE ('IF' <X-CLAUSE $('IF' X-CLAUSE)>
0011400-      ('ELSE' EXPRESSION .[IF,*4,*3,-[*2]-,*1] /.EMPTY
0011500-      .[IF,*3,*2,-[*1]-])/'ELSE' EXPRESSION
0011600-      .[IF,*3,*2,*1]/.EMPTY .[IF,*2,*1]);
0011700-X-CLAUSE = EXPRESSION 'THEN' SIMPLE-EXPRESSION;
```

Source

IF A < B THEN 2 ELSE B-6

Intermediate

(IF (LS A B) 2 (DIFFERENCE B 6))

Source

IF A < 0 THEN X IF A > 0 THEN Y ELSE Z

Intermediate

(IF (LS A 0) X (GR A 0) Y Z)

Source

```
IF A OR B THEN TRUE
```

Intermediate

```
(IF (OR A B) TRUE)
```

Source

```
IF A < 0 THEN 5 IF A > 0 THEN 7 IF A+B = 20 THEN 3 ELSE 21  ;
```

Intermediate

```
(IF (LS A 0) 5 (GR A 0) 7 (EQ (PLUS A B) 20) 3 21)
```

3.20 'FOR' VARIABLE EQUATIONS

```
0 010300-FOR-STATEMENT = 'FOR' VARIABLE ('←'/.EMPTY) <FOR-LIST> 'DO'
0010400-      STATEMENT .[FOR,*3,-[*2]-,*1];
0010500-FOR-LIST = FOR-ELEMENT $ (',' FOR-ELEMENT);
0010600-FOR-ELEMENT = <(↑'IN'/↑'ON') EXPRESSION TERM-ELEMENT> \
0010700-      '(' ')' .[NIL] \
0010800-      < (EXPRESSION\.EMPTY) (↑'STEP' EXPRESSION
0010900-      (↑'UNTIL' EXPRESSION/.EMPTY) /
0011000-      (↑'RESET' EXPRESSION/.EMPTY)) TERM-ELEMENT> \
0011100-      .EMPTY .[];
0011200-TERM-ELEMENT = (↑'WHILE'/↑'UNLESS') UNION/.EMPTY;
```

Source

```
FOR N ← 1 STEP 1 UNTIL K1, RESET B UNLESS R > 4 DO
BEGIN A(N) ← F(N); R←A(N)+3.5 END
```

Intermediate

```
(FOR N (1 STEP 1 UNTIL K1)
(RESET B UNLESS (GR R 4))
(BLOCK NIL (SET (A N) (F N)) (SET R (PLUS (A N) 3.5))))
```

Source

```
FOR L IN CAR H(W) WHILE ATOM L DO M←F(L) . G(L)
```

Intermediate

```
(FOR L (IN (CAR (H W)) WHILE (ATOM L))
     (SET M (CONS (F L)(G L))))
```

3.21 LABEL AND SIMPLE STATEMENT EQUATIONS

```
0009500-LABLE = IDENTIFIER ':';
0009600-SIMPLE-STATEMENT = ('GO' NAME-EXP .[GO,*1] /
0009700-      'RETURN' EXPRESSION .[RETURN,*1] /
0009800-      'TRY' STATEMENT ';' FULL-LOCATIVE ';' STATEMENT
0009900-      .[TRY,*3,*2,*1] /
0010000-      'CODE' '(' .[CODE, .[READ] ] ')' ) \
0010100-      LABLE SIMPLE-STATEMENT \
0010200-      SIMPLE-EXPRESSION;
```

Source	Intermediate
GO A(I)	(GO (A I))
RETURN -(X-Y)	(RETURN (MINUS (DIFFERENCE X Y)))
S1: S2: X ← Y	S1 S2 (SET X Y)
CAR A . B	(CONS (CAR A) B)

3.22 STATEMENT AND CONDITIONAL STATEMENT EQUATIONS

```
0008800-STATEMENT = COND-STATEMENT\SIMPLE-STATEMENT;
0008900
COND-STATEMENT = IF-STATEMENT/FOR-STATEMENT/LABLE COND-STATEMENT;
0009000-IF-STATEMENT = 'IF' C-CLAUSE ('IF' <C-CLAUSE $('IF' C-CLAUSE) ->
0009100-      ('ELSE' STATEMENT .[IF,*4,*3,-[*2]-,*1] /.EMPTY
0009200-      .[IF,*3,*2,-[*1]-])/'ELSE' STATEMENT
0009300-      .[IF,*3,*2,*1]/.EMPTY .[IF,*2,*1]);
0009400-C-CLAUSE = EXPRESSION 'THEN' SIMPLE-STATEMENT;
```

Source

```
IF ABLE < 0 THEN BEGIN X←ABLE; GO SLABLE END
  ELSE X(L) . Y(L)
```

Intermediate

```
(IF (LS ABLE 0) (BLOCK NIL (SET X ABLE)(GO SLABLE))
     (CONS (X L) (Y L)))
```

3.23 SWITCH DECLARATION EQUATION

```
0008500-SWITCH-DECL = 'SWITCH' IDENTIFIER '←' NAME-LIST ';'
0008600-      .[*2,SWITCH,-[*1]-];
0008700-NAME-LIST = < IDENTIFIER $ (',' IDENTIFIER) >;
```

Source

```
SWITCH EXAMPLE ← ST.LABLE1,ST.LABEL2;
```

Intermediate

```
(EXAMPLE SWITCH ST.LABEL1 ST.LABEL2)
```

3.24 BLOCK-DECLARATION EQUATION

```
0007300-BLOCK-DECL = <BLOCK-VAR-DECL $ BLOCK-VAR-DECL> PROCESS[];
0007400-BLOCK-VAR-DECL = (SWITCH-DECL\BLOCK-ARRAY-DECL\
0007500-      BLOCK-SIMPLE-DECL) ';' ;
0007600-BLOCK-ARRAY-DECL = <ARRAY-TYPE (MODE1/.EMPTY)> HOLD[]
0007700-      ARRAY-VAR-LIST;
0007800-BLOCK-SIMPLE-DECL = <TYPE (MODE1/.EMPTY)/MODE1 TYPE-OPTION>
0007900-      HOLD[] BLOCK-LIST;
0008000-BLOCK-LIST = BLOCK-VAR $ (',' BLOCK-VAR);
0008100-BLOCK-VAR = VARIABLE ('←' F-EXP .[*2,-[(DECLARE)]-,*1]/
0008200-      ↑'ASSIGNED' F-EXP .[*3,*2,*1] /
0008300-      .EMPTY .[*1,-[(DECLARE)]-]);
0008400-MODE1 = ↑'LOC' STORAGE-MODE/↑'FLUID' TRANSMISSION-MODE;
```

Source

```
REAL X,Y←3.26,PI←3.14159265; INTEGER A←1,B,C←762;
```

Intermediate

```
((X REAL)
  (Y REAL 3.26)
  (PI REAL 3.14159265) (A INTEGER 1) (B INTEGER) (C INTEGER 762))
```

Calls on LISP routines may be made from syntax equations where required for special actions not expressible in the meta language. This mechanism is used in declaration processing. In both the BLOCK-ARRAY-DECL and the BLOCK-SIMPLE-DECL, (see Syntax and Semantics of LISP IL, TM-2710/220/00) a routine called HOLD[] is invoked after the type of information in the declaration has been recognized and is sitting on top of the star stack. This type information

must be associated with each variable appearing in the remainder of the declaration. The routine HOLD[] stores the top element of the star stack into a variable called DECLARE. It also puts part of the declarative information in a variable called ARRAY-T for arrays. Wherever an identifier appears within .[] brackets, that identifier will appear in the corresponding position of the output list formed. However, if an identifier within brackets is also enclosed in parentheses (), the contents of the cell associated with that identifier will appear in the corresponding position of the resultant output list.

To illustrate these points, presented below is an example of the processing of a BLOCK-SIMPLE-DECL equation.

Input Tape	* Stack	Declare	Comments
REAL LOC X,Y,Z;	*=((REAL))		
REAL LOC X,Y,Z;	*=((LOC REAL))		
REAL LOC X,Y,Z;	*=((REAL LOC))		At the > in BLOCK-SIMPLE-DECL
REAL LOC X,Y,Z;	*=( )	(REAL LOC)	HOLD[] does this.
REAL LOC X,Y,Z;	*=(X)	(REAL LOC)	When variable is recognized in BLOCK-VAR
REAL LOC X,Y,Z;	*=((X REAL LOC))	(REAL LOC)	At line 83.00 note the -[ ]- remove the ( )
REAL LOC X,Y,Z;	*=(Y (X REAL LOC))	(REAL LOC)	from REAL LOC in the * Stack.
REAL LOC X,Y,Z;	*=((Y REAL LOC) (X REAL LOC))	(REAL LOC)	
REAL LOC X,Y,Z;	*=((Z REAL LOC) (Y REAL LOC) (X REAL LOC))	(REAL LOC)	

At the end of the definition of block-declaration, a routine called PROCESS [] is invoked. This routine searches the top element of the star stack and combines all the block-variable-declarations that refer to the same variable. An example is presented below:

Source

```
REAL FLUID X,Y; LOC A,Y; INTEGER L,M←5; OWN L;
```

Intermediate

Before PROCESS []

```
((X REAL FLUID) (Y REAL FLUID) (A LOC) (Y LOC) (L INTEGER) (M INTEGER 5) (L OWN))
```

After PROCESS []

```
((X REAL FLUID)
(Y REAL FLUID LOC) (A LOC) (L INTEGER OWN) (M INTEGER 5))
```

3.25 BLOCKS AND EXPRESSIONS

```
0006900-F-EXP = FUNCTIONAL\EXPRESSION;
0007000-EXPRESSION = BLOCK/COND-EXPRESSION/SIMPLE-EXPRESSION;
0007100-BLOCK = 'BEGIN' (BLOCK-DECL/.EMPTY .[])
0007200-      < $(-'END' STATEMENT (';'/.EMPTY)) > 'END'
0007210-      .[BLOCK,*2,-[*1]-];
```

Source

```
BEGIN REAL X,Y; SWITCH ABLE←S1,S2,S3; RETURN -(X-Y) END
```

Intermediate

```
(BLOCK ((X REAL) (Y REAL) (ABLE SWITCH S1 S2 S3))
  (RETURN (MINUS (DIFFERENCE X Y))))
```

Source

```
BEGIN INTEGER M,N; REAL X←3.7;
   A1: IF G1(N) THEN RETURN (M+N);
       M ← M+X;
       N ← N-X;
       X ← G2(X);
       GO A1
   END
```

Intermediate

```
(BLOCK ((M INTEGER) (N INTEGER) (X REAL 3.7))
 A1 (IF (G1 N) (RETURN (PLUS M N)))
 (SET M (PLUS M X)) (SET N (DIFFERENCE N X)) (SET X (G2 X)) (GO A1))
```

3.26 FUNCTION DECLARATIONS

```
0 0005700-FUNCTION-DECL = (VALUE-TYPE 'FUNCTION' VARIABLE .[*1,*1] /
0005800-      'FUNCTION' VARIABLE) PAR-NAME-LIST (';'/.EMPTY)
0005900-      PAR-DECL-LIST .[-[*2]-,-[*1]-] PROCESS[]
0006000-      <EXPRESSION/(';'/.EMPTY)> .[FUNCTION,*3,*2,-[*1]-];
0006100-PAR-NAME-LIST = '(' <I-PAR $ (',' VARIABLE .[*1])/.EMPTY> ')';
0006200-I-PAR = VARIABLE ('(' IDENTIFIER ')' .[*2,INDEF,*1]/
0006300-      .EMPTY .[*1] );
0006400-PAR-DECL-LIST = <PAR-DECL $ PAR-DECL /.EMPTY> ;
0 0006500-PAR-DECL = <TYPE (MODE1/.EMPTY)/MODE1 TYPE-OPTION>
0006600-      HOLD[] PAR-LIST ';' ;
0 0006700-PAR-LIST = VARIABLE .[*1,-[(DECLARE)]-] $ (',' VARIABLE
0006800-      .[*1,-[(DECLARE)]-]);
```

Source

```
FUNCTION PATH1 (WAY)
IF CAR WAY = FINISH THEN WAY ELSE
 BEGIN SYMBOL X,Y; FOR X IN GRAPH DO
   IF CAR X = CAR WAY
AND NOT MEMBER(CDR X,WAY) AND Y←PATH1(CDR X . WAY) THEN
   RETURN Y END  ;
```

Intermediate

```
(FUNCTION PATH1 (WAY)
 (IF (EQ (CAR WAY) FINISH)
  WAY (BLOCK ((X SYMBOL) (Y SYMBOL))
   (FOR X (IN GRAPH)
    (IF (AND (EQ (CAR X) (CAR WAY))
      (NOT (MEMBER (CDR X) WAY))
      (SET Y (PATH1 (CONS (CDR X) WAY)))) (RETURN Y))))))
```

Source

```
INTEGER FUNCTION SUMSQUARE(X(I))
    BEGIN INTEGER Y,J;
       FOR J←1 STEP 1 UNTIL I DO Y←Y+X(J)↑2;
       RETURN Y  END  ;
```

Intermediate

```
(FUNCTION (SUMSQUARE INTEGER)
   ((X INDEF I))
  (BLOCK ((Y INTEGER) (J INTEGER))
   (FOR J (1 STEP 1 UNTIL I) (SET Y (PLUS Y (EXPT (X J) 2))))
   (RETURN Y)))
```

Source

```
NOVALUE FUNCTION MATRIX.MULTIPLY(X,Y,Z,L,M,N) REAL ARRAY X,Y,Z;
     INTEGER L,M,N; LOC Z;
     BEGIN INTEGER I,J,K; Z←MAKEARRAY(L,N,'REAL);
          FOR I←1 STEP 1 UNTIL L DO
             FOR K←1 STEP 1 UNTIL N DO
                FOR J←1 STEP 1 UNTIL M DO
          Z(I,K) ← Z(I,K) + X(I,J) * Y(J,K) END  ;
```

Intermediate

```
(FUNCTION (MATRIX.MULTIPLY NOVALUE)
((X(ARRAY REAL)) (Y(ARRAY REAL))(Z(ARRAY REAL)LOC)
 (L INTEGER) (M INTEGER) (N INTEGER))
(BLOCK ((I INTEGER) (J INTEGER) (K INTEGER))
 (SET Z (MAKEARRAY L N (QUOTE REAL)))
 (FOR I (1 STEP 1 UNTIL L)
  (FOR K (1 STEP 1 UNTIL N)
   (FOR J (1 STEP 1 UNTIL M)
    (SET (Z I K) (PLUS (Z I K) (TIMES (X I J) (Y J K)))))))))
```

3.27 FREE DECLARATIONS

```
0001400-FREE-DECL-LIST = <FDL1> PROCESS[] .[DECLARE,-[*1]-];
0001500-FDL1 = FREE-DECL (FDL1 \.EMPTY);
0001600-FREE-DECL = ('DECLARE' (FREE-VAR-DECL/VARIABLE-LIST) /
0001700-      FREE-VAR-DECL) ';';
0001800-FREE-VAR-DECL = FREE-ARRAY-DECL\FREE-SIMPLE-DECL;
0001900-FREE-ARRAY-DECL = <ARRAY-TYPE (MODE/.EMPTY)> HOLD[]
0002000-      ARRAY-VAR-LIST;
0002100-ARRAY-VAR-LIST = ARRAY-VAR $ (',' ARRAY-VAR);
0002200-ARRAY-VAR = VARIABLE ('[' NUM-LIST ']' ('←' <ARRAY/FUNCTIONAL>
0002300-      .[*3,-[(DECLARE)]-,.[CREATE,-[*2]-,.[QUOTE,(ARRAY-T)]
0002400-       ,-[*1]-]] /
0002500-      .EMPTY .[*2,-[(DECLARE)]-,.[CREATE,-[*1]-,
0002600-       .[QUOTE,(ARRAY-T)]]]) /
0002700-      .EMPTY .[*1,-[(DECLARE)]-]);
0002800-NUM-LIST = <NUM $ (',' NUM)>;
0002900-NUM..  DGT $ DGT MAKEATOM[];
0003000-FREE-SIMPLE-DECL = <TYPE (MODE/.EMPTY)/MODE TYPE-OPTION>
0003100-      HOLD[] FREE-LIST;
0003200-FREE-LIST = FREE-VAR $ (',' FREE-VAR);
0003300-FREE-VAR = VARIABLE ('←' F-EXP .[*2,-[(DECLARE)]-,*1]/
0003400-      .EMPTY .[*1,-[(DECLARE)]-]);
0003500-SYNONYM-DECL = VARIABLE '==' VARIABLE .[*2,MEANS,*1];
0003600-TYPE-OPTION = TYPE/.EMPTY ;
0003700-TYPE = ARRAY-TYPE\FORMAL-TYPE\SIMPLE-TYPE;
0003800-SIMPLE-TYPE = ↑'REAL'/↑'INTEGER'/↑'SYMBOL'/
0003900-      ↑'BOOLEAN'/↑'OCTAL';
0004000-ARRAY-TYPE = (↑'FORMAL'/SIMPLE-TYPE) 'ARRAY' .[ARRAY,*1];
0004100-FORMAL-TYPE = (↑'NOVALUE'/↑'FORMAL'/SIMPLE-TYPE)
0004200-      'FORMAL' '(' <INDEF-PAR-TYPE PAR-TYPE-LIST> ')'
0004300-      .[FORMAL,*2,-[*1]-];
0004400-INDEF-PAR-TYPE = PARAMETER-TYPE 'INDEF' (',')/.EMPTY)
0004500-      .[*1,INDEF]\.EMPTY;
0004600-PAR-TYPE-LIST = PARAMETER-TYPE $(',' PARAMETER-TYPE)/.EMPTY;
0004700-PARAMETER-TYPE = F-TYPE ('LOC' .[*1,LOC]/.EMPTY) /
0004800-      ↑'LOC' (F-TYPE .[*1,*1]/.EMPTY);
0004900-TRANSMISSION-MODE = ↑'LOC'/.EMPTY;
0005000-F-TYPE = ↑'FORMAL'/SIMPLE-TYPE;
0005100-VALUE-TYPE = ↑'NOVALUE'/F-TYPE;
0005200-FREE-STORAGE-MODE = ↑'OWN'/STORAGE-MODE;
0005300-STORAGE-MODE = ↑'FLUID'/.EMPTY;
0005400-VARIABLE-LIST = '(' < VARIABLE $ (',' VARIABLE) > ')';
0005500-MODE = ↑'LOC' FREE-STORAGE-MODE/
0005600-      (↑'OWN'/↑'FLUID') TRANSMISSION-MODE;
```

Source

```
INTEGER ARRAY P[4]; BOOLEAN W,Y←TRUE,Z←FALSE; REAL A,B←3.14;
 SYMBOL D←CAR X . Y,E; OCTAL FLUID L,M←77Q3; INTEGER OWN Q←5E3;
```

Intermediate

```
(DECLARE (P (ARRAY INTEGER) (CREATE 4 (QUOTE INTEGER)))
   (W BOOLEAN)
   (Y BOOLEAN TRUE)
   (Z BOOLEAN FALSE)
   (A REAL)
   (B REAL 3.14)
   (D SYMBOL (CONS (CAR X) Y))
   (E SYMBOL)
   (L OCTAL FLUID) (M OCTAL FLUID 77Q3) (Q INTEGER OWN 5E3))
```

Source

```
BOOLEAN ARRAY I[4]←[BOOLEAN TRUE FALSE NIL ()];
REAL X,Y; INTEGER FLUID A,B; LOC X,A←L(M,N);
```

Intermediate

```
(DECLARE (I (ARRAY BOOLEAN) (CREATE 4 (QUOTE BOOLEAN) [TRUE FALSE NIL NIL]))
   (X REAL LOC)
   (Y REAL) (A INTEGER FLUID LOC (L M N)) (B INTEGER FLUID))
```

3.28 SECTION DECLARATION

```
0001100-SECTION-DECL = <TYPE-OPTION> 'SECTION' SECTION-NAME
0001200-      .[SECTION,*1,-[*1]-];
0001300-SECTION-NAME = ↑'NIL'/NAME-LIST;
```

Source	Intermediate
REAL SECTION X,Y,Z;	(SECTION (X Y Z) REAL)
SECTION A	(SECTION(A))

3.29 THE TOP LEVEL EQUATIONS OF LISP 2

```
0000100-.META(LISPII PROGRAM)
0000200-PROGRAM = $ (-'STOP' (DECLARATIVE\EXPRESSION) (';'/.EMPTY)
0000300-      COMPILE[]) 'STOP' .[STOP] COMPILE[];
0000400-DECLARATIVE = SECTION-DECL\FREE-DECL-LIST\FUNCTION-DECL\
0000500-      MACRO-DEF\INSTRUCTIONS-DEF\LAP-DEF\SYNONYM-DECL;
0000600-MACRO-DEF = ('SYMBOL'/.EMPTY) 'MACRO' VARIABLE '(' VARIABLE ')'
0000700-      EXPRESSION .[MACRO,*3, .[*2],*1];
0000800-INSTRUCTIONS-DEF = ('NOVALUE'/.EMPTY) 'INSTRUCTIONS' VARIABLE
0000900-      EXPRESSION .[INSTRUCTIONS,.[*2,NOVALUE],NIL,*1];
0001000-LAP-DEF = 'LAP' '(' .[LAP, .[READ] ] ')';
```

At the top level (1.00), the first identifier shows the name of the translator that we are producing, namely LISP II. That is, the LISP 2 syntax translator is called by typing LISP II (X,Y). (The X is the input file and the Y corresponds to the output file.) The second identifier, namely PROGRAM, indicates that the starting point of the translator is where the translator looks for PROGRAM that is the top level syntax equation.

PROGRAM is a sequence of declaratives or expressions followed by an optional ;. After each of these declaratives or expressions are recognized, the corresponding intermediate language is written on the output file. The word STOP results in a call on (STOP) being produced that is used to terminate compilation when the intermediate language is given to the compiler. The final example shows a trivial program which finds 3! iteratively and recursively.

```
LISPII (TTY TTY)
INTEGER FUNCTION FACTORIAL(N);
     BEGIN INTEGER X,I ← 1;
     FOR X ← 1 STEP 1 UNTIL N DO
          I ← I * X; RETURN I END;
(FUNCTION (FACTORIAL INTEGER)
((N))
(BLOCK ((X INTEGER) (I INTEGER 1))
(FOR X (1 STEP 1 UNTIL N) (SET I (TIMES I X))) (RETURN I)))
INTEGER FUNCTION RFACTORIAL(N);
     IF N = ∅ THEN 1 ELSE N*RFACTORIAL(N-1);
(FUNCTION (RFACTORIAL INTEGER)
((N)) (IF (EQ N ∅) 1 (TIMES N (RFACTORIAL (DIFFERENCE N 1)))))
FACTORIAL(3); RFACTORIAL(3); STOP
(FACTORIAL 3)
(RFACTORIAL 3)
(STOP)
```

Throughout the syntax equations the slash is used to separate alternatives. The careful reader will have noticed that two types of slashes are used to separate alternatives: the normal slash (/) and the reverse slash (\). The reverse slash indicates that the state of the machine, namely the input tape pointer, the star stack, etc. must be saved since backup may be required for a correct parse. This backup will occur when the first element or elements to be recognized of a number of alternatives legally start with the same construct. This ability to indicate when backup should take place in the definition of a language is called the controlled backup feature. The syntax equation below (Figure 1) illustrates this point. If A fails, C is tried, if B fails, G is tried, etc.

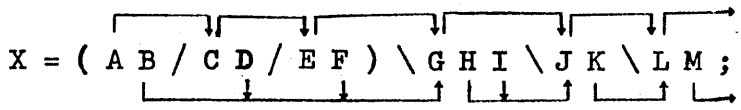

Figure 1. Controlled Backup Feature

APPENDIX A

The Complete Specifications for LISP 2 (SL) to LISP 2 (IL)

```
0000100-.META(LISPII PROGRAM)
0000200-PROGRAM = $ (-'STOP' (DECLARATIVE\EXPRESSION) (';'/.EMPTY)
0000300-      COMPILE[]) 'STOP' .[STOP] COMPILE[];
0000400-DECLARATIVE = SECTION-DECL\FREE-DECL-LIST\FUNCTION-DECL\
0000500-      MACRO-DEF\INSTRUCTIONS-DEF\LAP-DEF\SYNONYM-DECL;
0000600-MACRO-DEF = ('SYMBOL'/.EMPTY) 'MACRO' VARIABLE '(' VARIABLE ')'
0000700-      EXPRESSION .[MACRO,*3, .[*2],*1];
0000800-INSTRUCTIONS-DEF = ('NOVALUE'/.EMPTY) 'INSTRUCTIONS' VARIABLE
0000900-      EXPRESSION .[INSTRUCTIONS,.[*2,NOVALUE],NIL,*1];
0001000-LAP-DEF = 'LAP' '(' .[LAP, .[READ] ] ')';
0001100-SECTION-DECL = <TYPE-OPTION> 'SECTION' SECTION-NAME
0001200-      .[SECTION,*1,-[*1]-];
0001300-SECTION-NAME = ↑'NIL'/NAME-LIST;
0001400-FREE-DECL-LIST = <FDL1> PROCESS[] .[DECLARE,-[*1]-];
0001500-FDL1 = FREE-DECL (FDL1 \.EMPTY);
0001600-FREE-DECL = ('DECLARE' (FREE-VAR-DECL/VARIABLE-LIST) /
0001700-      FREE-VAR-DECL) ';';
0001800-FREE-VAR-DECL = FREE-ARRAY-DECL\FREE-SIMPLE-DECL;
0001900-FREE-ARRAY-DECL = <ARRAY-TYPE (MODE/.EMPTY)> HOLD[]
0002000-      ARRAY-VAR-LIST;
0002100-ARRAY-VAR-LIST = ARRAY-VAR $ (',' ARRAY-VAR);
0002200-ARRAY-VAR = VARIABLE ('[' NUM-LIST ']' ('←' <ARRAY/FUNCTIONAL>
0002300-      .[*3,-[(DECLARE)]-,.[CREATE,-[*2]-,.[QUOTE,(ARRAY-T)]
0002400-       ,-[*1]-]] /
0002500-      .EMPTY .[*2,-[(DECLARE)]-,.[CREATE,-[*1]-,
0002600-       .[QUOTE,(ARRAY-T)]]]) /
0002700-      .EMPTY .[*1,-[(DECLARE)]-]);
0002800-NUM-LIST = <NUM $ (',' NUM)>;
0002900-NUM..  DGT $ DGT MAKEATOM[];
0003000-FREE-SIMPLE-DECL = <TYPE (MODE/.EMPTY)/MODE TYPE-OPTION>
0003100-      HOLD[] FREE-LIST;
0003200-FREE-LIST = FREE-VAR $ (',' FREE-VAR);
0003300-FREE-VAR = VARIABLE ('←' F-EXP .[*2,-[(DECLARE)]-,*1]/
0003400-      .EMPTY .[*1,-[(DECLARE)]-]);
0003500-SYNONYM-DECL = VARIABLE '==' VARIABLE .[*2,MEANS,*1];
0003600-TYPE-OPTION = TYPE/.EMPTY ;
0003700-TYPE = ARRAY-TYPE\FORMAL-TYPE\SIMPLE-TYPE;
0003800-SIMPLE-TYPE = ↑'REAL'/↑'INTEGER'/↑'SYMBOL'/
0003900-      ↑'BOOLEAN'/↑'OCTAL';
0004000-ARRAY-TYPE = (↑'FORMAL'/SIMPLE-TYPE) 'ARRAY' .[ARRAY,*1];
0004100-FORMAL-TYPE = (↑'NOVALUE'/↑'FORMAL'/SIMPLE-TYPE)
0004200-      'FORMAL' '(' <INDEF-PAR-TYPE PAR-TYPE-LIST> ')'
0004300-      .[FORMAL,*2,-[*1]-];
0004400-INDEF-PAR-TYPE = PARAMETER-TYPE 'INDEF' (',' /.EMPTY)
0004500-      .[*1,INDEF]\.EMPTY;
0004600-PAR-TYPE-LIST = PARAMETER-TYPE $(',' PARAMETER-TYPE)/.EMPTY;
0004700-PARAMETER-TYPE = F-TYPE ('LOC' .[*1,LOC]/.EMPTY) /
0004800-      ↑'LOC' (F-TYPE .[*1,*1]/.EMPTY);
0004900-TRANSMISSION-MODE = ↑'LOC'/.EMPTY;
```

```
0005000-F-TYPE = ↑'FORMAL'/SIMPLE-TYPE;
0005100-VALUE-TYPE = ↑'NOVALUE'/F-TYPE;
0005200-FREE-STORAGE-MODE = ↑'OWN'/STORAGE-MODE;
0005300-STORAGE-MODE = ↑'FLUID'/.EMPTY;
0005400-VARIABLE-LIST = '(' < VARIABLE $ (',' VARIABLE) > ')';
0005500-MODE = ↑'LOC' FREE-STORAGE-MODE/
0005600-      (↑'OWN'/↑'FLUID') TRANSMISSION-MODE;
0005700-FUNCTION-DECL = (VALUE-TYPE 'FUNCTION' VARIABLE .[*1,*1] /
0005800-      'FUNCTION' VARIABLE) PAR-NAME-LIST (';'/.EMPTY)
0005900-      PAR-DECL-LIST .[-[*2]-,-[*1]-] PROCESS[]
0006000-      <EXPRESSION/(';'/.EMPTY)> .[FUNCTION,*3,*2,-[*1]-];
0006100-PAR-NAME-LIST = '(' <I-PAR $ (',' VARIABLE .[*1])/.EMPTY> ')';
0006200-I-PAR = VARIABLE ('(' IDENTIFIER ')' .[*2,INDEF,*1]/
0006300-      .EMPTY .[*1] );
0006400-PAR-DECL-LIST = <PAR-DECL $ PAR-DECL /.EMPTY> ;
0006500-PAR-DECL = <TYPE (MODE1/.EMPTY)/MODE1 TYPE-OPTION>
0006600-      HOLD[] PAR-LIST ';' ;
0006700-PAR-LIST = VARIABLE .[*1,-[(DECLARE)]-] $ (',' VARIABLE
0006800-      .[*1,-[(DECLARE)]-]);
0006900-F-EXP = FUNCTIONAL\EXPRESSION;
0007000-EXPRESSION = BLOCK/COND-EXPRESSION/SIMPLE-EXPRESSION;
0007100-BLOCK = 'BEGIN' (BLOCK-DECL/.EMPTY .[])
0007200-      < $(-'END' STATEMENT (';'/.EMPTY)) > 'END'
0007210-      .[BLOCK,*2,-[*1]-];
0007300-BLOCK-DECL = <BLOCK-VAR-DECL $ BLOCK-VAR-DECL> PROCESS[];
0007400-BLOCK-VAR-DECL = (SWITCH-DECL\BLOCK-ARRAY-DECL\
0007500-      BLOCK-SIMPLE-DECL) ';' ;
0007600-BLOCK-ARRAY-DECL = <ARRAY-TYPE (MODE1/.EMPTY)> HOLD[]
0007700-      ARRAY-VAR-LIST;
0007800-BLOCK-SIMPLE-DECL = <TYPE (MODE1/.EMPTY)/MODE1 TYPE-OPTION>
0007900-      HOLD[] BLOCK-LIST;
0008000-BLOCK-LIST = BLOCK-VAR $ (',' BLOCK-VAR);
0008100-BLOCK-VAR = VARIABLE ('←' F-EXP .[*2,-[(DECLARE)]-,*1]/
0008200-      ↑'ASSIGNED' F-EXP .[*3,*2,*1] /
0008300-      .EMPTY .[*1,-[(DECLARE)]-]);
0008400-MODE1 = ↑'LOC' STORAGE-MODE/↑'FLUID' TRANSMISSION-MODE;
0008500-SWITCH-DECL = 'SWITCH' IDENTIFIER '←' NAME-LIST ';'
0008600-      .[*2,SWITCH,-[*1]-];
0008700-NAME-LIST = < IDENTIFIER $ (',' IDENTIFIER) >;
0008800-STATEMENT = COND-STATEMENT\SIMPLE-STATEMENT;
0008900
COND-STATEMENT = IF-STATEMENT/FOR-STATEMENT/LABLE COND-STATEMENT;
0009000-IF-STATEMENT = 'IF' C-CLAUSE ('IF' <C-CLAUSE $('IF' C-CLAUSE) >
0009100-      ('ELSE' STATEMENT .[IF,*4,*3,-[*2]-,*1] /.EMPTY
0009200-      .[IF,*3,*2,-[*1]-])/'ELSE' STATEMENT
0009300-      .[IF,*3,*2,*1]/.EMPTY .[IF,*2,*1]);
0009400-C-CLAUSE = EXPRESSION 'THEN' SIMPLE-STATEMENT;
0009500-LABLE = IDENTIFIER ':';
```

```
0009600-SIMPLE-STATEMENT = ('GO' NAME-EXP .[GO,*1] /
0009700-      'RETURN' EXPRESSION .[RETURN,*1] /
0009800-      'TRY' STATEMENT ';' FULL-LOCATIVE ';' STATEMENT
0009900-      .[TRY,*3,*2,*1] /
0010000-      'CODE' '(' .[CODE, .[READ] ] ')' ) \
0010100-      LABLE SIMPLE-STATEMENT \
0010200-      SIMPLE-EXPRESSION;
0010300-FOR-STATEMENT = 'FOR' VARIABLE ('←'/.EMPTY) <FOR-LIST> 'DO'
0010400-      STATEMENT .[FOR,*3,-[*2]-,*1];
0010500-FOR-LIST = FOR-ELEMENT $ (',' FOR-ELEMENT);
0010600-FOR-ELEMENT = <(↑'IN'/↑'ON') EXPRESSION TERM-ELEMENT> \
0010700-      '(' ')' .[NIL] \
0010800-      < (EXPRESSION\.EMPTY) (↑'STEP' EXPRESSION
0010900-      (↑'UNTIL' EXPRESSION/.EMPTY) /
0011000-      (↑'RESET' EXPRESSION/.EMPTY)) TERM-ELEMENT> \
0011100-      .EMPTY .[];
0011200-TERM-ELEMENT = (↑'WHILE'/↑'UNLESS') UNION/.EMPTY;
0011300
COND-EXPRESSION = 'IF' X-CLAUSE ('IF' <X-CLAUSE $('IF' X-CLAUSE)>
0011400-      ('ELSE' EXPRESSION .[IF,*4,*3,-[*2]-,*1] /.EMPTY
0011500-      .[IF,*3,*2,-[*1]-])/'ELSE' EXPRESSION
0011600-      .[IF,*3,*2,*1]/.EMPTY .[IF,*2,*1]);
0011700-X-CLAUSE = EXPRESSION 'THEN' SIMPLE-EXPRESSION;
0011800-FUNCTIONAL = 'FUNCTIONAL' '(' EXPRESSION ';' FUNARG-LIST ')'
0011900-      .[FUNCTIONAL,*2,-[*1]-] /
0012000-      (VALUE-TYPE 'FUNCTION' (VARIABLE\FALSIE) .[*1,*1]/
0012100-      'FUNCTION' (VARIABLE\FALSIE)) PAR-NAME-LIST (';'/.EMPTY)
0012200-      PAR-DECL-LIST .[-[*2]-,-[*1]-] PROCESS[] '(' EXPRESSION
0012300-      (';' FUNARG-LIST .[FUNCTION,*4,*3,*2,-[*1]-] /
0012400-      .EMPTY .[FUNCTION,*3,*2,*1] ) ')';
0012500-FUNARG-LIST = <VARIABLE $ (',' VARIABLE)\.EMPTY>;
0012600-SIMPLE-EXPRESSION = UNION ('. ' SIMPLE-EXPRESSION
0012700-      .[CONS,*2,*1] /.EMPTY );
0012800-UNION = INTERSECTION ('OR' <INTERSECTION $ ('OR' INTERSECTION)>
0012900-      .[OR,*2,-[*1]-]/.EMPTY );
0013000-INTERSECTION = NEGATION ('AND' <NEGATION $ ('AND' NEGATION)>
0013100-      .[AND,*2,-[*1]-]/.EMPTY );
0013200-NEGATION = NEGATIVE/RELATION;
0013300-NEGATIVE = (↑'NOT'/↑'NULL') NEGATION .[*2,*1];
0013400-RELATION = SUM $(('/=' ↑NQ/'=' ↑EQ/'<=' ↑LQ/'<' ↑LS/
0013500-      '>=' ↑GQ/'>' ↑GR) SUM .[*2,*2,*1] );
0013600-SUM = SUM1 ('+' < SUM1 $ ('+' SUM1) >
0013700-      .[PLUS,*2,-[*1]-] /.EMPTY);
0013800-SUM1 = FACTOR $ ('-' FACTOR .[DIFFERENCE,*2,*1] );
0013900-FACTOR = FACTOR1 ('*' < FACTOR1 $ ('*' FACTOR1) >
0014000-      .[TIMES,*2,-[*1]-] /.EMPTY);
0014100-FACTOR1 = PRIMARY $(('/' ↑QUOTIENT PRIMARY\'\' ↑REMAINDER
0014200-      PRIMARY\'-:' ↑IQUOTIENT PRIMARY) .[*2,*2,*1] );
0014300-PRIMARY = '+' PRIMARY/'-' PRIMARY .[MINUS,*1] /
0014400-      ↑'ATOM' PRIMARY .[*2,*1] /
0014500-      UNIT ('↑' PRIMARY .[EXPT,*2,*1] /.EMPTY);
```

```
0014600-UNIT = NEGATIVE/CONSTANT/'(' EXPRESSION ')' /BLOCK/
0014700-      LOCATIVE ('←' F-EXP .[SET,*2,*1]/.EMPTY);
0014800-LOCATIVE = LIST-LOCATIVE\WORD-LOCATIVE;
0014900-LIST-LOCATIVE = (CR-EXP\↑'PROP') (LOCATIVE\'(' EXPRESSION ')'
0015000-      ) .[*2,*1];
0015100-CR-EXP.. +'C' (+'A'/+'D') $(+'A'/+'D') +'R'
0015200-      MAKEATOM[];
0015300-WORD-LOCATIVE = ↑'BIT' '(' EXPRESSION ',' EXPRESSION
0015400-      ',' WORD-LOCATIVE ')' .[*4,*3,*2,*1] /
0015500-      ↑'CORE' '(' EXPRESSION ')' .[*2,*1] /
0015600-      FULL-LOCATIVE;
0015700-FULL-LOCATIVE = NAME-EXP ('←←' FULL-LOCATIVE .[LOCSET,*2,*1]/
0015800-      .EMPTY );
0015900-NAME-EXP = VARIABLE ('(' CALL-PART ')' .[*2,-[*1]-]/.EMPTY);
0016000-VARIABLE = IDENTIFIER ('$' (IDENTIFIER .[EXTERNAL,*2,*1]/
0016100-      '$' .[EXTERNAL,*1]) /.EMPTY );
0016200-CALL-PART = < F-EXP $ (',' F-EXP)\.EMPTY >;
0016300-CONSTANT = SIMPLE-ATOM/QUOTED-EXP;
0016400-QUOTED-EXP = '''' S-EXPRESSION .[QUOTE,*1];
0016500-S-EXPRESSION = ATØM/LIST;
0016600-LIST = '(' < S-EXPRESSION $ S-EXPRESSION
0016700-      (+'. ' S-EXPRESSION/.EMPTY) > ')' ;
0016800-ATØM = SIMPLE-ATOM\IDENTIFIER\C-ATOM\<FUNCTION-SPEC>\<ARRAY>;
0016900-C-ATOM.. +'''' ANY MAKEATOM[];
0017000-SIMPLE-ATOM = BOOLEAN\NUMBER\STRING;
0017100-IDENTIFIER.. LET $ (LET/DGT/+'.') MAKEATOM[] RESERVED[] /
0017200-      +'%' SIMPLE-STRING MAKEATOM[];
0017300-STRING.. SIMPLE-STRING $ (SIMPLE-STRING SCONS[*2,*1])
0017400-      MAKEATOM[];
0017500-SIMPLE-STRING = +'#' $ (-'#' (-'''' ANY/DELETEX ANY)) +'#';
0017600-BOOLEAN = ↑'TRUE'/FALSIE;
0017700-FALSIE = ('FALSE'/'NIL'/'(' ')') ↑NIL;
0017800-NUMBER.. ('+'/+'-'/.EMPTY) (REAL\OCTAL\INTEGER) MAKEATOM[];
0017900-REAL = REAL-MANTISSA REAL-EXPONENT;
0018000-SIGN = '+'/+'-'/.EMPTY;
0018100-REAL-MANTISSA = +'.' UNSIGNED-DECIMAL/UNSIGNED-DECIMAL +'.'
0018200-      (UNSIGNED-DECIMAL/.EMPTY);
0018300-REAL-EXPONENT = +'E' SIGN UNSIGNED-DECIMAL /.EMPTY;
0018400-INTEGER = UNSIGNED-DECIMAL DECIMAL-EXPONENT;
0018500-DECIMAL-EXPONENT = +'E' (UNSIGNED-DECIMAL/.EMPTY)/.EMPTY;
0018600-UNSIGNED-DECIMAL = DGT $ DGT ;
0018700-OCTAL = OCT $ OCT +'Q' (UNSIGNED-DECIMAL/.EMPTY);
0018800-LET = ISIT[LETTER,T];  DGT = ISIT[DIGIT,T];
0018900-OCT = ISIT[8,T];
0019000-FUNCTION-SPEC = ↑'[' 'FUNCTION' '(' VARIABLE
0019100-      VALUE-TYPE ↑')' INDEF-PAR-TYPE $PARAMETER-TYPE
0019200-      ↑']' ;
0019300-ARRAY = ↑'[' (NUMBER-ARRAY/BOOLEAN-ARRAY/FORMAL-ARRAY/
0019400-      SYMBOL-ARRAY) ↑']' ;
0019500-NUMBER-TYPE = ↑'REAL'/↑'INTEGER'/↑'OCTAL';
```

```
0019600-NUMBER-ARRAY = NUMBER-TYPE ($ NUMBER/$ NUMERIC-ROW);
0019700-NUMERIC-ROW = ↑'[' (NUMBER $ NUMBER /
0019800-      NUMERIC-ROW $ NUMERIC-ROW ) ↑']' ;
0019900-BOOLEAN-ARRAY = ↑'BOOLEAN' ($BOOLEAN/$ BOOLEAN-ROW);
0020000-BOOLEAN-ROW = ↑'['(BOOLEAN $ BOOLEAN /
0020100-      BOOLEAN-ROW $ BOOLEAN-ROW ) ↑']' ;
0020200-FORMAL-ARRAY = ↑'FORMAL' $ (+'*' FUNCTION-SPEC)
0020300-      $ FORMAL-ROW ↑']' ;
0020400-FORMAL-ROW = ↑'['(↑'*' FUNCTION-SPEC $(↑'*' FUNCTION-SPEC)
0020500-      /FORMAL-ROW $ FORMAL-ROW) ↑']' ;
0020600-SYMBOL-ARRAY = ↑'SYMBOL' ($ SYMBOL-ELEMENT/$ SYMBOL-ROW);
0020700-SYMBOL-ELEMENT = SIMPLE-ATOM/IDENTIFIER/LIST
0020800-      / $ (FUNCTION-SPEC/ARRAY) ;
0020900-SYMBOL-ROW = ↑'['(SYMBOL-ELEMENT $ SYMBOL-ELEMENT /
0021000-      SYMBOL-ROW $ SYMBOL-ROW) ↑']' ;
0021100-.FINISH
0021200-.STOP
```