60496400

CONTROL DATA CORPORATION

SYMPL VERSION 1 REFERENCE MANUAL

CONTROL DATA®
CYBER 170 SERIES
CYBER 70 SERIES
7000 SERIES
6000 SERIES COMPUTER SYSTEMS

REVISION RECORD	
REVISION	DESCRIPTION
A	Original printing.
(11-1-75)	
Publication No. 60496400	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

PREFACE

The SYMPL Version 1.1 compiler makes efficient use of storage during compilation and generation of machine language instructions. Implementation of this system provides simultaneous compilation of several programs, utilizing the operating system's multiprogramming features. The SYMPL compiler operates under the control of:

> NOS 1 operating system for the CONTROL DATA® CYBER 170, CYBER 70 Models 72, 73, 74, and 6000 Series Computer Systems.
>
> NOS/BE 1 operating systems for the Control Data CYBER 70 Models 72, 73, 74, and 6000 Series Computer Systems.
>
> SCOPE 2 operating system for the Control Data CYBER 76 and 7600 Computer Systems.

This reference manual presents the semantics and rules for writing programs in the SYMPL language; it also includes sufficient information to prepare, compile, and execute such programs. Syntax, or the structure of SYMPL word forms and their mutual relations, appears in the appendix section.

It is assumed that the reader has some knowledge of the NOS 1 and NOS/BE 1 operating systems and Control Data computer systems.

The following manuals contain additional information:

Publication	Publication Number
NOS 1.0 Operating System Reference Manual, Volume 1	60435400
NOS/BE 1 Operating System Reference Manual	60493800
INTERCOM 4 Reference Manual	60494600

CONTENTS

INTRODUCTION 1

SYMPL (System Programming Language) was designed for use by system programmers in writing compilers and system software. It omits many facilities typically found in higher-level languages, being unencumbered with those extra features normally included for scientific and commercial applications programming. It is intended to facilitate compilation, and it places minimum restrictions on allowable optimizations.

OPERATING SYSTEM INTERFACE

The compiler is designed to run under control of standard operating systems. Compilation is requested by a control statement specifying the name SYMPL. This call results in the loading and execution of the subprogram SYMPL which controls the compilation process. The compiler obtains the control statement parameters from the operating system.

SYMPL OVERVIEW

SYMPL, which is a procedure oriented language, is similar to JOVIAL, which was derived from ALGOL-58 (the 1958 version of the International Algorithmic Language, as described in the December 1958 issue of the **Communications of the ACM**).

CONCISE LANGUAGE

SYMPL is a readable and concise programming language, utilizing self-explanatory English words and the familiar notations of algebra and logic. In addition, SYMPL has no format restrictions; therefore, the programmer may intermix comments among the symbols of a program and define notational additions to the language. Also, a SYMPL program may serve as its own documentation, allowing easy maintenance and revision.

CODING CONVENTIONS

Coding conventions for SYMPL are less restrictive then most languages. The source program is considered simply as a stream of characters: card or line boundaries are ignored. Significant columns are 1-72; 73 on are not interpreted. For purposes of source information, the compiler assumes column 72 is adjacent to column 1 of the next card or line. Names, constants, operators, or any SYMPL symbol, can be broken across cards or lines.

LANGUAGE ELEMENTS

The organization of SYMPL language elements is shown in figure 1-1.

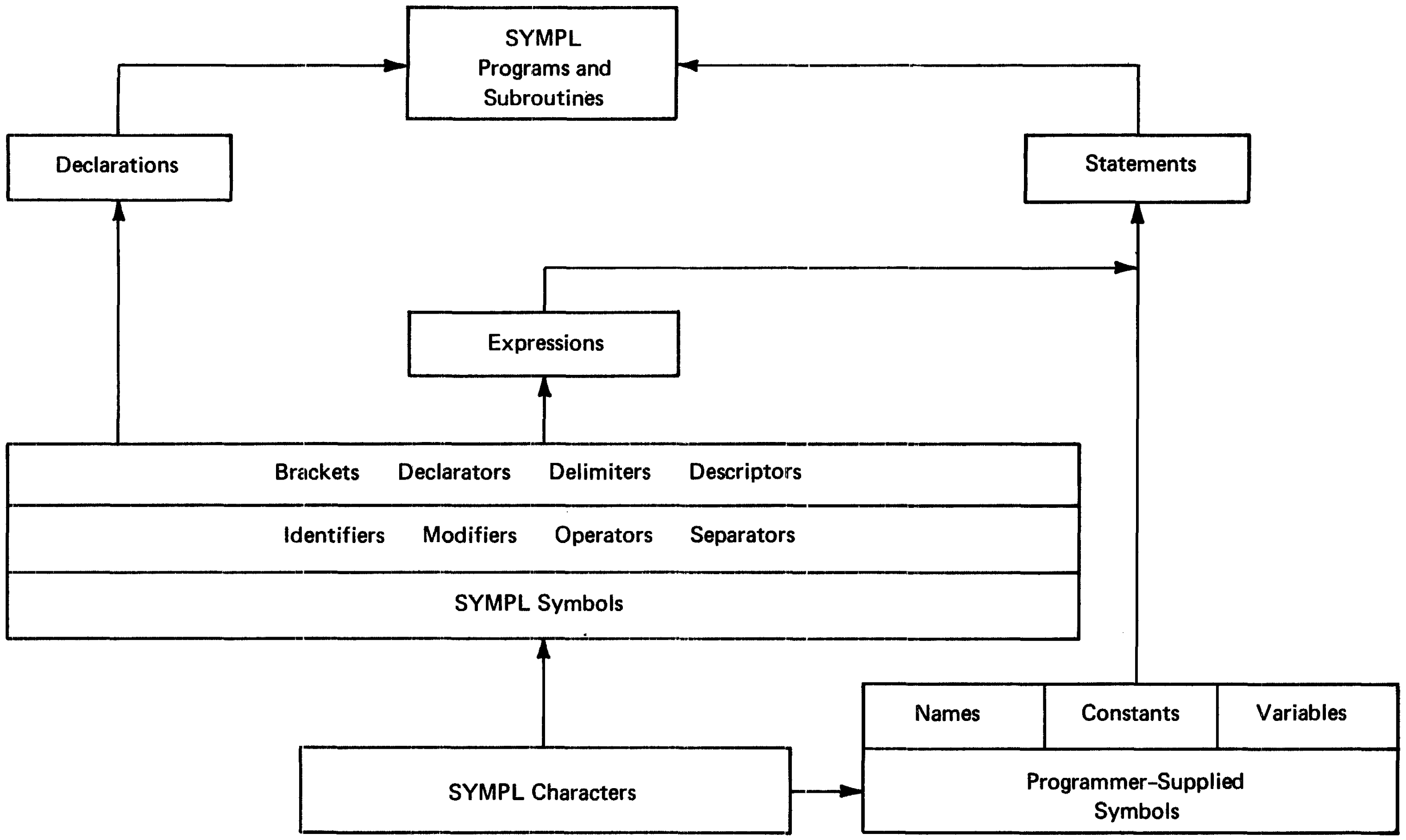


Figure 1-1. Elements of the SYMPL Language

BASIC NOTATION AND ELEMENTS 2

CHARACTERS

Characters are the basic elements of the SYMPL language and are commonly available on standard keypunch and typewriter equipment.

SYMPL Characters:

27 Letters:	A-Z and $
10 Numerals:	0-9
18 Marks:	* / + - = < > ( ) [ ] ″ ′ . , ; : blank

The double prime or quote (″) is represented by the hardware mark (≡) and the 0/8/6 multipunch (Hollerith 026) or the 3/8 multipunch (Hollerith 029); the prime (′) is represented by the hardware mark (≠) and the 4/8 multipunch (Hollerith 026) or a 8/7 multipunch (Hollerith 029).

For a detailed description of characters, numerals, and marks as a function of Hollerith Code, Display Code, and External BCD, see Appendix A.

BLANK SPACE AND COMMENTARY

Characters classified as marks serve as delimiting characters. For example, any one of them may be used to delimit character sequences forming identifiers. Normally, the concatentation of two nondelimiting characters does not have the same effect as their occurrence separated by delimiters (XY is not the same as X.Y or X Y). Since the blank space is an element within the set of marks, it is classified as a delimiter and its use is significant. Whenever one blank is required as a delimiter, any number of blanks will suffice; wherever a nonblank delimiter is required, it may be embedded in a sequence of blanks of arbitrary length.

A comment is an arbitrary string of characters which can be inserted between or within SYMPL statements and declarations; comments are enclosed with a pair of double primes and must not contain a double prime or a semicolon. For example:

Correct	"THIS IS A COMMENT"	or	≡ THIS IS A COMMENT ≡
Incorrect	""SUCCESSFUL ABORT"		
Incorrect	"INCORRECT COMMENT;"		

The hardware mark (equivalence symbol) ≡ will be used throughout much of this manual, primarily to avoid confusion with two consecutive single quotes (′ ′).

Comments may be of any length and may appear wherever it is legal to write a blank, with the following exceptions:

Within a status constant

Within a comment

After the name in a DEF declaration

IDENTIFIERS

Identifiers are arbitrary names used to label the various elements in a SYMPL program; they express reserved SYMPL words and name programmer defined entities.

The 52 reserved identifiers which represent SYMPL words must not be used for entity names. A complete list of the 52 reserved words appears in appendix D.

Restrictions imposed on the choice of identifiers:

First character must be a letter

Not more than 12 characters in length

Cannot include marks

Examples of valid identifiers:

XYZ

$A

UR12

Examples of character sequences which are not identifiers:

2X	Begins with a digit
'Z	Begins with a prime
X'Y	Embedded prime
IDGRTRTHAN12 CHARS	Exceeds 12 characters

Examples of reserved identifiers which may not be used as user identifiers:

ITEM

TEST

TERM

IF

ARITHMETIC OPERATORS

Arithmetic operators denote basic arithmetic operations and the following Boolean operations:

Symbol	Meaning	Example
+	Addition	AA + BB
+	Unary plus	
-	Subtraction	XX - YY
-	Unary minus	
*	Multiplication	DIAM * 3.14
/	Division	IN/CM
**	Exponentiation	VOL ** 3
LAN	Logical and	
LNO	Logical not	
LOR	Logical or	
LXR	Logical exclusive or	
LIM	Logical imply	
LQV	Logical equivalent	

RELATIONAL OPERATORS

Relational operators denote numerical relationship between quantities.

Symbol	Meaning	Example
EQ	Is equal to	AA EQ BB
GR	Is greater than	DISTANCE GR 500.0
GQ	Is greater than or equal to	INCOME GQ 10000
LQ	Is less than or equal to	LIMIT LQ 50
LS	Is less than	LIAB LS ASSETS
NQ	Is not equal to	VELOCITY NQ 70

BOOLEAN OPERATORS

Boolean operators denote the three basic operations of Boolean algebra.

Symbol	Meaning	Example
AND	Conjunction	DAY AND NIGHT
OR	Union	FRIEND OR FOE
NOT	Negation	NOT CLEAR

DATA 3

CONSTANTS

Each of the five types of constants is a sequence of characters which defines its own value; constants represent specific, fixed values that do not change during program execution. The constants are: Boolean, character, integer, real, and status.

BOOLEAN CONSTANT

Boolean constants, TRUE or FALSE, represent the two elements of Boolean algebra; generally a Boolean value is used to represent TRUE or FALSE, ON or OFF, YES or NO. The value for TRUE is 1; value for FALSE is 0.

CHARACTER CONSTANT

Character constants represent alphanumeric data.

Format:

    'character-string'

Any character may be included in the 'character-string' except a single prime, which must be represented by two consecutive primes. The maximum length of a character constant is 240 characters.

Examples:

```
'THIS IS A CHARACTER CONSTANT'
'THIS ONE''S TRICKY'
```

INTEGER CONSTANT

Integer constants may be expressed as: decimal integer, hexadecimal constant, octal constant, or status function. The format determines the manner in which the value is represented in the computer memory.

During execution, the maximum allowable value is $2^{48}-1$ when an integer constant is converted to real. If the result is greater than $2^{48}-1$, bits 48 through 58 will be ignored and errors may result. The maximum value of the operands and the result of integer multiplication or division must be less than $2^{48}-1$. High order bits will be lost if the value is larger, but no diagnostic is provided.

Decimal Integer

A decimal integer is a string of decimal digits. Embedded blanks are not allowed in a decimal integer; it may contain up to 18 decimal digits and must not exceed $2^{59}-1$ in value.

Hexadecimal Constant

Hexadecimal constants may be used to specify binary bit patterns. When hexadecimal constants appear in integer formulas, the value must not exceed $2^{59}-1$.

Format:

X 'hexadecimal integer'
A hexadecimal integer is a string of hexadecimal digits (0-9 and the letters A-F).

Embedded blanks may be included; however, they are ignored during compilation.

Examples:

Hexadecimal Constant	**Decimal Equivalent**	**Bit Pattern**
X'F'	15	1111
X'4BC'	1212	010010111100

Octal Constant

Octal constants may be used to specify binary bit patterns. When octal constants appear in integer formulas, the value must not exceed $2^{59}-1$.

Format:

O 'octal integer'

An octal integer is a string of up to 20 octal digits (0-7).

Embedded blanks may be included in octal constants; however, they are ignored during compilation.

Examples:

```
O'56'       O'777776'

O'7'        O'55232522202211230555'
```

Status Function

The status function is a special form of integer constant discussed under **STATUS DECLARATIONS.**

STATUS CONSTANT

Status constants are discussed under **STATUS DECLARATIONS.**

REAL CONSTANT

A real constant is represented by a string of decimal digits including a decimal point and an optional exponent representing multiplication by a power of 10. The exponent is written as the letter D or E followed by an optional plus or minus sign and a decimal integer. No embedded blanks are allowed; D and E are semantically equivalent.

A real constant is represented within the computer as a normalized floating point number. The magnitude of a real constant may be in the range 10^{-293} to 10^{322}, with up to 15 digits of accuracy.

Examples:

```
3.E1        31.41592E-01       .5

6.4D+35     3.141592E+279      0.0
```

ITEM DECLARATIONS

SYMPL items are named, value representative, entities declared by item declarations; and they gain value by arithmetic replacement. The item is the basic unit used to describe the structure of data to be manipulated. Programmer defined items must be declared prior to any reference to them.

Item type may be any of the following: Boolean, character, integer, real, status, or unsigned integer.

The general format of the ITEM declaration is:

ITEM name$_1$ type preset, name$_2$ type preset . . . ;

Item name is a programmer supplied identifier and item type is one of the six forms shown below.

Preset specifies an optional initial value to be assigned to the item; its format is given below. Each item description of an item declaration defines one item name of the given type:

Item Character	Type	Format
B	Boolean	B
I	Integer	I
R	Real	R
U	Unsigned integer	U
C	Character	C (length)
S	Status	S: status list name

length is an integer constant, not greater than 240, specifying the number of characters in the item, and status list name is the name of a STATUS list from which the item is to assume values (see STATUS DECLARATION).

If type is omitted, integer type is assumed.

Examples:

ITEM X R;	≡DEFINES X AS TYPE REAL≡
ITEM Y,Z C(10);	≡DEFINES Y AS INTEGER ITEM AND Z AS CHARACTER ITEM OF SIZE 10≡
ITEM STAT S:SNAME;	≡ITEM STAT WILL ASSUME VALUES ASSOCIATED WITH STATUS LIST NAME≡

ITEM PRESETS

An item may be assigned an initial value with a preset.

Format of item preset:

= ± constant	the sign is optional

The type of the constants used in item presets need not be commensurate with the type of the item; if they are not, however, the resulting item value may not be meaningful, since the value of the constant is inserted without conversion into a field size determined by the item characteristics. Character constants will be left justified within the item and either right filled with blanks or right truncated, as necessary.

Examples:

Item Declarations	**Preset Item Declarations**
ITEM BOOVAL B;	ITEM BOOVAL B = 1;
ITEM CHAR C(10);	ITEM CHAR C(10) = '3.14159';
ITEM GAMMA I;	ITEM GAMA I = 380;
ITEM ZETA R;	ITEM ZETA R = 1.414;
ITEM SAILBOAT S:LIST;	ITEM SAILBOAT S:LISTA = S'TORO':

STATUS LIST DECLARATIONS

The status list declares a list of identifiers to which the compiler assigns monotonically increasing values, beginning with zero. Its purpose is to allow mnemonic reference to certain variables of small integer value.

Format:

STATUS Lname name_1, name_2, . . . ;

Lname identifies the status list.

name_1,name_2, . . .	Called status values are assigned monotonically increasing integer values starting at zero.

Examples:

```
STATUS WORDS BEGIN, END, TERM, STATUS, WORDS;

STATUS LIST1 POOR, FAIR, GOOD, EXCELLENT;

STATUS LIST2 OK, NOGOOD;

STATUS SIMLAR2 NOTGOOD,OK; ≡OK MAY BE IN TWO LISTS≡

STATUS ALPHA   A, B, C, D, E, F, G, H, I, J, K, L, M,
               N, O, P, Q, R, S, T, U, V, W, X, Y, Z;

STATUS COLOR RED, ORANGE, YELLOW, BLUE, GREEN;
```

Status value identifiers, unlike other identifiers, need not be unique within a program since the status list with which they are associated can always be determined from context; status values are used in the following ways:

STATUS FUNCTION

Format:

status-list-name 'status value'

A status function may be used anywhere an integer constant may be. It defines a unique integer constant value.

Examples:

In the previous examples

```
X=COLOR'ORANGE';          ≡ EQUIVALENT TO X=1 ≡

ITEM JJ I = LIST1'GOOD' ;
```

STATUS CONSTANT

Format:

S'status value'

Since a status constant does not define a unique value, it must be used only in conjunction with a status item which refers to the status list of which it is a member. A status constant may be used in expressions and as presets.

Examples:

Referring to the status list examples:

```
ITEM VAL S: WORDS = S'END' ;

ITEM LETTER S: ALPHA ;
```

```
LETTER = S'B' ;

IF LETTER EQ S'Q' THEN . . .
```

STATUS SWITCH

See section 5 for a discussion of switches.

ARRAYS

Certain classes of problems may require variables to be arranged in terms of one or more dimensions. Such an arrangement of item-like elements is called an array. An array may be viewed as a rectangular assortment of entries, each composed of one particular instance of each item comprising the array. The array concept refers collectively to several array items, without reference to value. The dimensionality of arrays is unrestricted. A typical array is the two dimensional array or matrix where each element resides in a particular row or column.

Example:

	column			
	0	1	2	3
0	4	0	7	-8
row 1	23	-9	11	6
2	-7	14	-2	77

In this array the value 77 resides in row 2, column 3. Because there are 3 rows and 4 columns, this array has the dimensions 3 by 4.

DECLARATION

An array declaration consists of a header and one or more array item declarations; the header format follows:

Format:

ARRAY name [$l_1:u_1, l_2:u_2 \ldots$] $\begin{Bmatrix} P \\ S \end{Bmatrix}$ $\underline{\text{(entry-size)}}$;

Name is the array identifier; it may be omitted unless the ARRAY declaration appears in a BASED, XDEF, or XREF declaration.

[$l_1:u_1, l_2:u_2 \ldots$] specifies dimensions of the array

l_i lower bound
u_i upper bound $\Big\}$ integers, modulo 2^{18}

The number of pairs ($l_i:u_i$) is the dimension of the array and $l_i \leq u_i$.

If l_i is omitted it is assumed zero and the colon is also omitted.

If the bounds list is omitted [0:0] is assumed.

The number of entries in the array is:

$$(u_1-l_1+1)\ (u_2-l_2+1)\ldots(u_n-l_n+1)$$

where the number of words reserved for the array is:

$$(u_1-l_1+1)\ (u_2-l_2+1)\ldots(u_n-l_n+1)\ (\underline{\text{entry-size}})$$

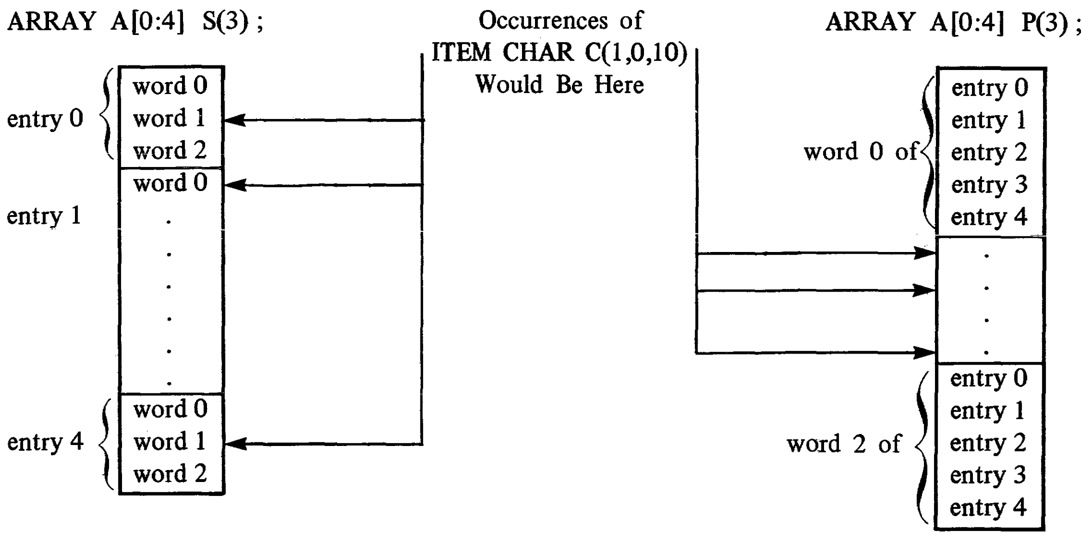
$\left\{\begin{matrix}P\\S\end{matrix}\right\}$ signifies that either P or S is to be chosen. P specifies parallel allocation; S serial allocation; if neither is selected, P is assumed.

entry-size is an unsigned integer specifying the number of words in an array entry; if omitted, 1 is assumed and the parentheses are also omitted. Each entry may consist of one or more array items as defined below.

An array entry exists corresponding to each unique set of defined subscripts.

In serial allocation, the words of each entry are allocated contiguously. In parallel allocation, the first words of each entry are allocated contiguously, followed by the second words, and so on. A form of dynamic array allocation is therefore possible with serial allocation only.

Example:



ARRAY ITEM DECLARATION

The array declaration header is followed by one or more array item declarations. If more than one appears, the series must be contained in a BEGIN, END bracket.

Format:

ITEM $name_1$ type (ep,fbit,size) preset,$name_2$ type (ep,fbit,size) preset, . . . ;

$name_i$ is the array item identifier.

type is as defined for items – one of B, I, U, C, R, S: status list

if type is omitted integer is assumed.

ARRAY STORAGE AND ADDRESSING

At compilation time, an array is allocated the following amount of storage:

$[(u_1 - l_1 + 1)*(u_2 - l_2 + 1)* \ldots *(u_n - l_n + 1)] * (\text{entry-size})$

(ep,fbit,size)

entry position (ep)	Word number in which the high-order bit of the item occurs; ep is the word number starting from 0, expressed as an integer constant.
first bit (fbit)	Bit number within the word (expressed as an integer constant) starting at the most significant bit of the word, with 0.
	If fbit is omitted, zero is assumed by default; if fbit and ep are omitted, they are both assumed to be zero.
size	Item length. For B type items, size is in bits and default size is one bit.
	For C type items, size, in bytes, consists of 6 contiguous bits with a first bit number of 0,6,12, . . . 54. Default size is one byte.
	For I, R, S, and U type items, the size unit is one bit and the default length is one word.

If the entire descriptor is omitted, the defaults are chosen as above.

If one argument is used: (ep); if two arguments are used: ep and fbit.

Preset allows initial values to be assigned to each instance of the array item.

RESTRICTIONS

B,I,R,S, and U type items must be contained entirely within one entry word and may not cross word boundaries.

C type items must be byte aligned and the length must be $\leq$ 240 bytes.

R type items may be declared to be less than one word. No special handling of the exponent will be provided by the compiler, however. Results from R type items which are less than one word will be meaningless.

The following chart summarizes array item allocation restrictions:

Type	fbit Alignment	Length Restriction	May Cross Words	Length Unit	Default Length
I,U	bit	word	no	bit	word
R	word	word	no	bit	word
B	bit	word	no	bit	bit
C	byte	240 bytes	yes	byte	byte
S	bit	word	no	bit	word

Given

```
ARRAY A[ℓ1:u1,ℓ2:u2, . . . ,ℓn:un] P/S(n);
    BEGIN
        ITEM AI U (ep);
        ITEM AL;
    END
```

the location of the array-item $AI[\ell_1,\ell_2, \ldots \ell_n]$ with respect to the location of its array name, is given by:

array-item address

= array-name-address + ep layout = S

= array-name-address + $[ep*(u_1-l_1+1)*(u_2-l_2+1)* \ldots]$ layout = P

Where array-name-address is the address of item AI[0,0, . . . 0] (even if the zeroth element does not exist).

For a three-dimension array, the relative location of A[i,j,k] with respect to $A[\ell_1,\ell_2,\ell_3]$ is given by:

location (A[i,j,k])

= location $(A[\ell_1,\ell_2,\ell_3])$ + (x + L *(y + M*(z)))*(entry-size)

where:

$x = i - \ell_1$

$y = j - \ell_2$

$z = k - \ell_3$

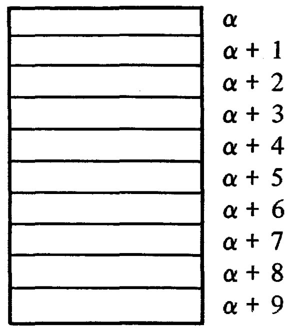
$L = u_1 - \ell_1 + 1$

$M = u_2 - \ell_2 + 1.$

For a parallel layout array and entry-size of 0, subscripts are calculated assuming that the entry-size = 1.

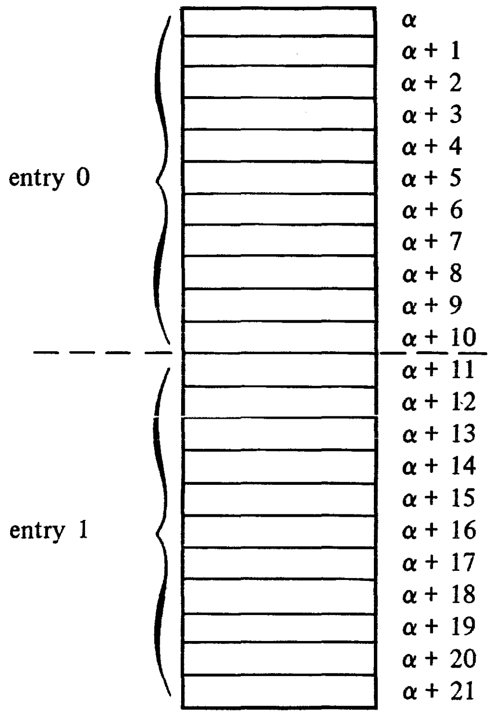
Examples of parallel layout arrays:

ARRAY ARY1[0:10];



Array length
= 11 * 1 = 11

ARRAY ARY2[0:10] P(2);

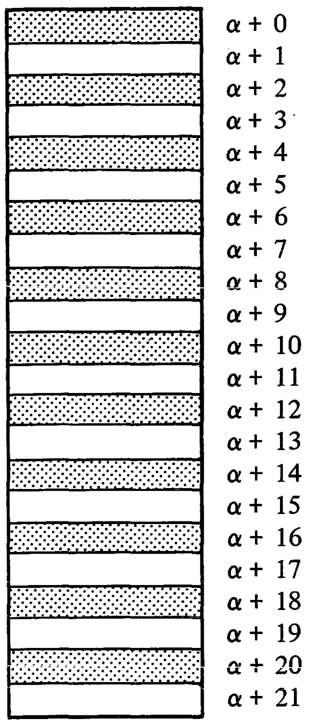


Array length
= 11 * 2 = 22

In ARY2, all ITEMS whose ep=0 will be found above the dotted line; those with ep=1 will be found below the dotted line. When ep>1, the ITEM will be defined outside the table limits. No range checks are made on subscripts or ep's.

Examples of serial layout arrays:

```
ARRAY ARY3[0:10] S(2);
```



Array length
= 11 * 2 = 22

In this array, all ITEMS whose ep=0 will be found at locations α + x where x is even; those with ep=1 will be found at locations α + y where y is odd. If ep is outside the proper range, the item will be defined at locations which are not normal. For example:

```
ITEM AA I(2,0,60);
```

is equivalent to

```
ITEM AB I(0,0,60);
```

except that

```
AA[s] = AB [s + 2)
```

In this illustration, the darkened areas indicate entries of ep=0, clear areas are entries of ep=1.

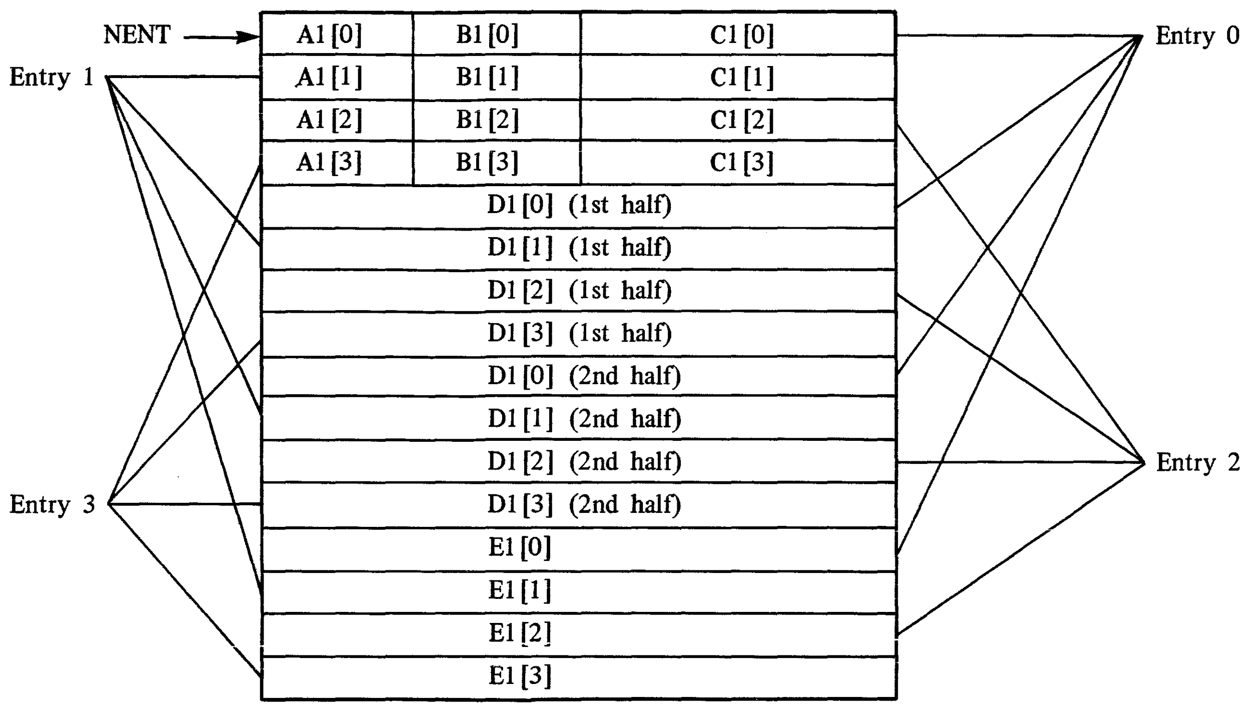
Example:

The core allocation for the table is:

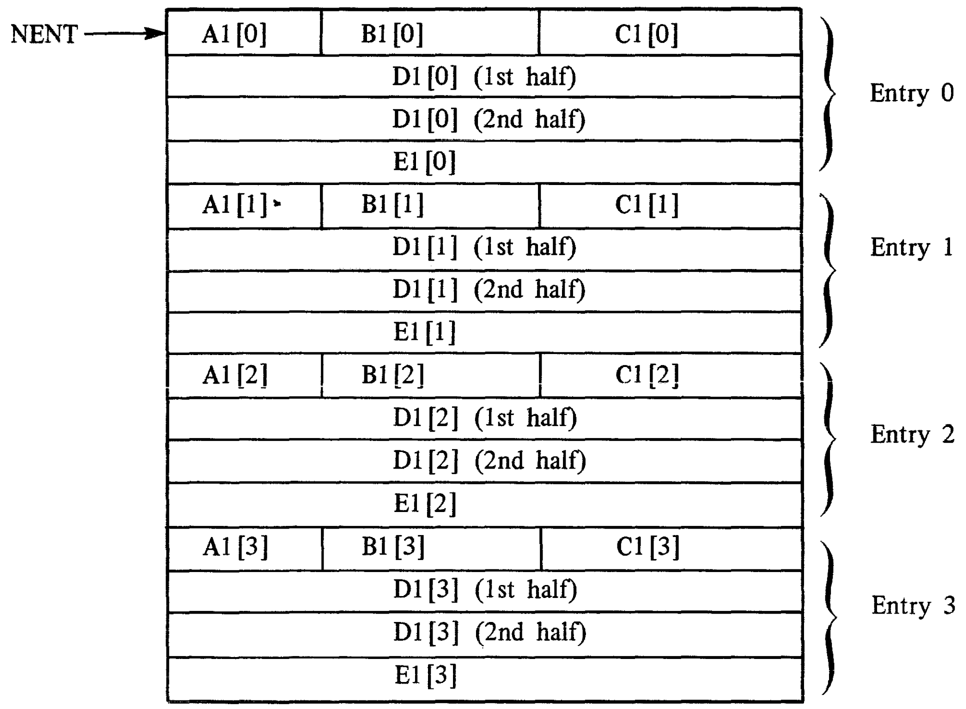
```
ARRAY NENT [0:8] P(4);
ITEM
    A1 I(0,0,15),
    B1 U(0,15,15),
    C1 U(0,30,30),
    D1 C(1,0,20),
    E1 R(3,0,60);
```

The same table with declaration S(4) instead of P(4) is shown in the following two illustrations:

PARALLEL ARRAY STRUCTURE



SERIAL ARRAY STRUCTURE



A serial array can be extended beyond its declared bounds, overlaying any variable allocated immediately after NENT. If this array is at the end of blank common, the array may extend into available high core. The array element A1[4], therefore, is in the central memory word beyond E1[3].

Extension of a parallel array beyond its declared bounds causes the elements of different entries to overlap each other. The array element A1[4] is, in fact, the top 15 bits of element D1[0]. The elements of the entry with the largest offset, in this case E1, may be extended in a manner similar to that for serial arrays. The array element E1[4] exists in the central memory word beyond E1[3].

This is true for access to elements within the array bounds if the offset word size of an item is greater than the entry size. For example, if an additional element was declared in the above example as:

```
    F1(4,0,60),
```

By arranging the proper lower and upper bounds, array-sizes, and ep's for various ARRAY's and ITEM's, many ITEMS may be forced to OVERLAY (as in JOVIAL) or become EQUIVALENT (as in FORTRAN) to other ITEMS. Furthermore, by using an array-size of 0, no storage is used but many addresses may be computed through the relationship of the lower/upper bounds and the ep's.

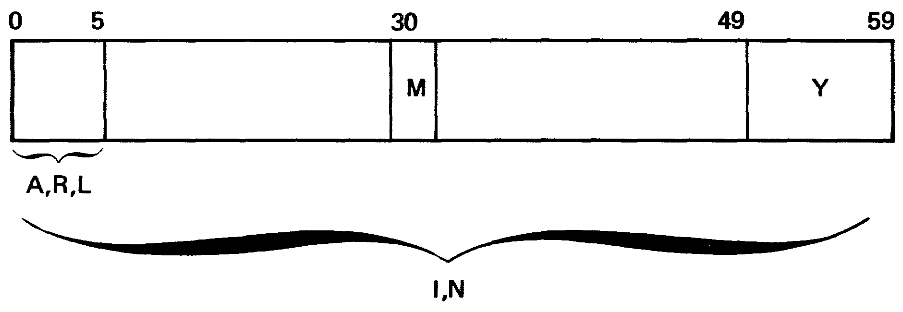
These practices should be avoided (or used with care). They can result in data structures that depend upon allocation by a specific loader, as well as interdependencies between tables not immediately apparent.

Some other examples:

```
ARRAY RAY[0:9] S(3);                          Resultant structure of array:
            ITEM  X  R  (0,0),
                                                   X[0]
                  Y  R  (1,0),                     Y[0]
                                                   Z[0]
                  Z  R  (2,0);                     X[1]
                                                   Y[1]
                                                   Z[1]
                                                    |
                                                    v
                                                   X[9]
                                                   Y[9]
                                                   Z[9]
```

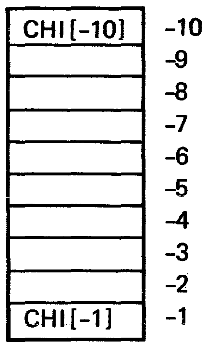
```
ARRAY PARALLEL [0];

            ITEM  M  B  (0,30,1),
                  A  I  (0,0,6),
                  R  C  (0,0,1),
                  I  I  (0,0,60),
                  L  C  (0),
                  Y  B  (0,50,10),
                  N  R  (0,0);
```

Resultant structure of above array:



```
ARRAY SIGMA [-10:-1];
    ITEM CHI I(0,0,60);
```

Resultant structure of above array:



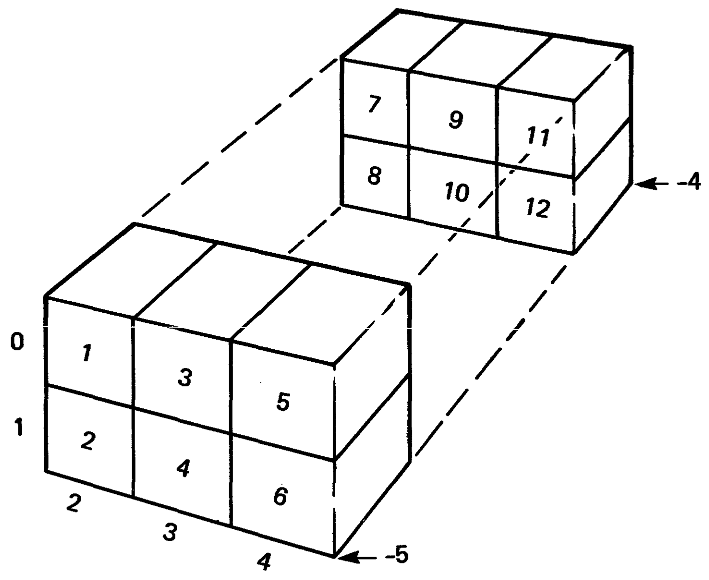
In this negatively subscripted parallel array, CHI[-10] is the first word of the array.

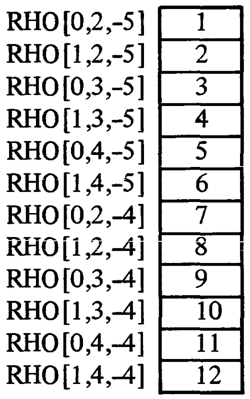
Array items are allocated in column order; that is, the leftmost subscript varies most rapidly.

For storage allocation consider the following multi-dimensional array:

ARRAY RHO[0:1,2:4,-5:-4];

Resultant structure of above array is:





RHO[0,2,-5]	1
RHO[1,2,-5]	2
RHO[0,3,-5]	3
RHO[1,3,-5]	4
RHO[0,4,-5]	5
RHO[1,4,-5]	6
RHO[0,2,-4]	7
RHO[1,2,-4]	8
RHO[0,3,-4]	9
RHO[1,3,-4]	10
RHO[0,4,-4]	11
RHO[1,4,-4]	12

Given the following two dimensional array:

```
ARRAY PSI[1:3, 0:3] (2);                 ARRAY PSI[1:3,0:3] (2);
     BEGIN                               BEGIN
          ITEM X,Y(1);          or            ITEM X I(0,0,60);
     END                                      ITEM Y I(1,0,60);
                                         END
```

The preceding two declarations are identical. The allocation of the elements depends on the specification of Parallel or Serial (P or S) in the array declaration (see page 3-7). In this example, the allocation is parallel by default. Compare the following allocation for Parallel and Serial arrays defined as above in other respects.

Parallel	Serial
X[1,0]	X[1,0]
X[2,0]	Y[1,0]
X[3,0]	X[2,0]
X[1,1]	Y[2,0]
X[2,1]	X[3,0]
X[3,1]	Y[3,0]
X[1,2]	X[1,1]
X[2,2]	Y[1,1]
X[3,2]	X[2,1]
X[1,3]	Y[2,1]
X[2,3]	X[3,1]
X[3,3]	Y[3,1]
Y[1,0]	X[1,2]
Y[2,0]	Y[1,2]
Y[3,0]	X[2,2]
Y[1,1]	Y[2,2]
Y[2,1]	X[3,2]
Y[3,1]	Y[3,2]
Y[1,2]	X[1,3]
Y[2,2]	Y[1,3]
Y[3,2]	X[2,3]
Y[1,3]	Y[2,3]
Y[2,3]	X[3,3]
Y[3,3]	Y[3,3]

PRESET VALUES

To specify a set of initial values for an array item, an array preset is appended to the array item declaration. Basically, it is a set of constant values, arranged in a list, with the same order as the allocation order of different instances of the items in storage. The list is presented in sections enclosed in square brackets, and nested to the depth of the number of dimensions in the array. An N dimensional array at the first level of nesting, has as many sections as the Nth dimension of the array. Each of these sections has as many sections as the N-1st dimension, etc. At the deepest level, each section has as many values as the first dimension of the array. Each section at the first level contains values for the instances of the array item with the same rightmost subscript; the subscript associated with each section varying from the lower bound at the left to the upper bound at the right. Each section of the second level contains values for those instances with the same rightmost two subscripts, etc. The outermost section is appended to the array item declaration with an equals sign.

Example of a preset parallel array:

```
    ARRAY OMEGA [0:1,0:2];
    ITEM MU     I(0,0) =[[ 1,2][ 3,4][5,6]];
```

is equivalent to:

```
    ARRAY OMEGA [0:1,0:2];
    ITEM MU I (0,0) ;
    MU[0,0] = 1;
    MU[1,0] = 2;
    MU[0,1] = 3;
    MU[1,1] = 4;
    MU[0,2] = 5;
    MU[1,2] = 6;
```

The constant list need not specify an initial value for every element. The values given are used to set elements starting with the first instance of the item. Rules for vacuous conditions are:

Null values are indicated by adjacent commas

Trailing null values are omitted

Null brackets are left empty

Thus, the following two array presets are equivalent:

[[[, ,2] [,1,]] [[, ,] [3,4,5]] [[, ,] [, ,]]]

is equivalent to

[[[, ,2] [,1]] [[ ] [3,4,5]] [ ]]

Repetition of values may be created by bracketing a list of values with parentheses and a count.

Thus,

3(2,1) is equivalent to 2,1,2,1,2,1

and

2(2(0,2)) is equivalent to 0,2,0,2,0,2,0,2

Repetition of bracketed sections is indicated by placing a count outside the bracket.

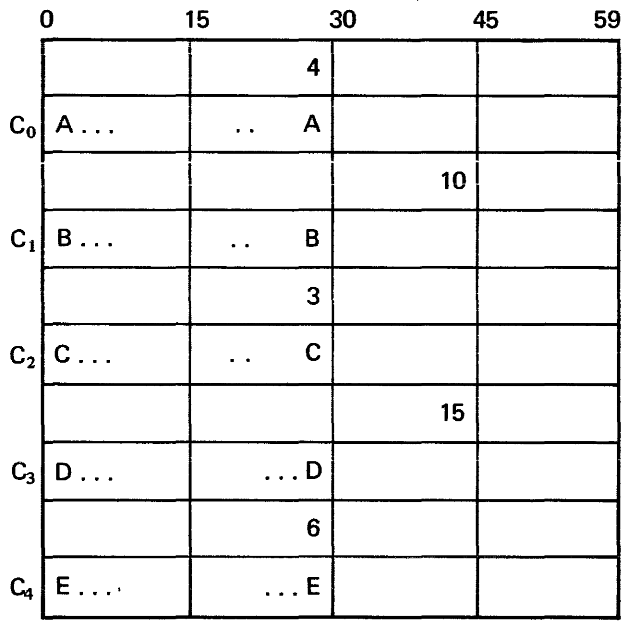
Thus,

2[[1,3] [2(2)]] is equivalent to [[1,3] [2,2]] [[1,3] [2,2]]

Further examples of array presets:

```
ARRAY TENWORD [0:4] S(2);
BEGIN
ITEM A I(0,0,30)  = [4,,3,,6];
ITEM B I(0,0,45)  = [,10,,15];
ITEM C C(1,0,5)  = ['AAAAA','BBBBB','CCCCC','DDDDD','EEEEE'];
END
```

The following allocation emerges from the above coding:



Multi-dimension arrays are preset using nested brackets as illustrated in the following example:

```
ARRAY XYZ [0:2,3:5,-4:-2];
BEGIN
        ITEM P I(0,00,60)= [3[3[3(4)]]];
END
```

```
ARRAY OUTRAY [ 1:10] S(3) ;
BEGIN
     ITEM OUTITEM C(0,00,07)= ['POS=','MAX=','BLK=' ];
     ITEM OUTSIZE I(0,42,18)= [ 10(4) ];
     ITEM OUTLO  I(1,00,12)= [ 1,1,,,,1 ];
     ITEM OUTHI  I(1,12,48)= [ 16383,8388607,,,,16777214 ];
     ITEM OUTALPHA C(1,00,08)= [,,'RUN','FVU','AE',,,'XBNC' ];
     ITEM OUTLEN I(2,00,12)= [6,7,1,1,1,8,9,1,3,10];
     ITEM OUTBIT I(2,12,12)= [45,21,12,,,,,,11];
     ITEM OUTNUM I(2,24,12)= [ 15,24,6,,,,,,1 ];
     ITEM OUTADD I(2,36,12)= [ 4,4,4,,,,,,4 ]:
END
```

ARRAY REFERENCE SUBSCRIPTS

To indicate a particular entry of an array, or a particular instance of an array item, a subscript is appended to the array item name.

A subscript list for an array reference will contain as many arithmetic expressions as there are array dimensions. Each arithmetic expression is evaluated as an integer value following the rules of type conversion, and the resultant integer (modulo 2^{18}) specifies the entry referenced.

If the natural type of the arithmetic expressions used as subscripts is other than integer, the expressions will be converted to integer mode.

Examples of array references:

```
ARRAY REF [0:1,0:2];

ITEM B  I(0,0) ;

B[1,1]

B[X+Y,1]

B[B[1,B[0,0]],B[1,B[X,1]]]
```

BASED ARRAYS AND THE P FUNCTION

A based array is one for which no storage is allocated by the compiler; however, the compiler does create a specific pointer variable, compiled with an undefined value. Reference to based arrays will be compiled using the pointer variable which is assumed to contain the defined array address. No implicit mechanism is provided to obtain space and set the pointer variable for based arrays; pointer values must be set explicitly by the programmer.

The based array name is declared in a based declaration, and reference is made to based arrays in the same manner as for non-based arrays.

Format:

BASED array-dec

or

BASED BEGIN array-dec array-dec . . . END

array-dec is an array declaration

P FUNCTION

Reference is made to the pointer variable using the intrinsic pointer or p function.

Format:

P<name>

name is the name of a based array.

Examples:

```
BASED ARRAY  AA[0:9 ];

BEGIN
     ITEM XX;
END

P<AA>=NXTAV ;

      FOR I=O STEP 1 UNTIL 9

      DO XX[I] = 9 -I;

PROC  GET(A);

BEGIN BASED ARRAY A[ 0: 999] ;

     BEGIN ITEM AA R(0,0,60); END
```

```
XREF PROC REAO;
ARRAY A1[0:999];
     BEGIN ITEM X R; END
ARRAY A2[0:999] ;
     BEGIN ITEM Y R; END
     ITEM GATE B ;
IF NOT GATE THEN P<A> = LOC(A1);
     ELSE P<A> = LOC(A2);
READ(A); ≡PASS THE LOCATION OF A≡
RETURN;
END ≡PROC GET≡
```

Using based arrays, the programmer may impose a structure any place in memory; however, no storage is allocated at compile time. At run-time, the programmer must define explicitly the location of the based array.

Example:

```
BASED ARRAY PRESET [99] ;
     ITEM WORD I(0,0,60);
     •
     •
     •
          ≡NOW SET THE ARRAY≡
P<PRESET> = LOC(A);
     FOR I=O STEP 1 UNTIL 99
     DC WORD[I] = 0;
          ≡LOC(A) COULD HAVE BEEN PASSEC TO MAIN
          PROGRAM BY ANOTHER PRCCEDURE≡
```

Based arrays should be used when the programmer does not know prior to run-time where the array is to be located. The array may be in a blank common area or in a labeled common area. For example: Use based arrays, for a symbol table with variable length entries when it is not known where an entry begins in blank common.

Based arrays allow access to absolute location within the program field length. In the example P<PRESET> = 0; WORD [1] is word one of the program field length.

By setting the pointer of a based array to the location of a label or procedure, the machine code at that location may be accessed.

EXPRESSIONS 4

An expression is a rule for computing a value. The values of the operands comprising the expression are combined according to the rules of the language to form a single value. Constants, simple items, subscripted array items, and function references are all expressions. Further, if x_1 and x_2 are expressions and op_1 and op_2 are unary and binary operators, respectively, then op_1 x_1 and x_1 op_2 x_2 are expressions.

Operators with higher precedence are evaluated before those with lower precedence, otherwise, expressions are evaluated from left to right. Parentheses may be used to change the order of evaluation. If x is an expression, (x) and ((x)) are also expressions. In evaluating A+B*C the multiplication is performed first. If the addition is to be performed first the expression is written as (A+B) *C.

ARITHMETIC EXPRESSIONS

An arithmetic expression containing only numeric operands and arithmetic operators has a numeric value. Numeric operands include constants, items, subscripted array items, and functions of type integer, real, or status.

The arithmetic operators are listed below in order of precedence (highest to lowest).

( )	Parentheses, beginning with the innermost pair
**	Exponentiation
*/	Multiplication and division, from left to right
+ -	Unary plus, minus
+ -	Addition and subtraction, from left to right
LNO	Logical no (complement)
LAN	Logical and
LOR	Logical or (inclusive)
LXR	Logical or (exclusive)
LIM	Logical imply
LQV	Logical equivalent

SYMPL has no implicit multiplication features in which algebraic multiplication can be indicated by X(Y) or 3X. Such multiplication in SYMPL must be explicit: XX * YY and 3 * XX.

Omission of an operator, as for implied multiplication (X) (Y), for instance, is not valid and results in a compiler diagnostic. Also, division by zero is undefined.

All function references and exponentiation operations which are not evaluated in-line are evaluated prior to other operations.

When writing an integer expression, it is important to remember not only the left-to-right scanning process but also if dividing an integer quantity by an integer quantity yields a remainder the result will be truncated; thus 11/3 = 3.

An array element name (a subscripted variable) used in an expression requires the evaluation of its subscript. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by the evaluation of the actual arguments or subscripts.

The operators LNO, LAN, LOR, LXR, LIM, and LQV form the bit-by-bit complement, product, etc. of the operands. The Glossary defines the operation of the individual operators.

The following examples are valid expressions:

```
A              - (C+DELTA*AERO)

3.14159        TEMP+V[M,MAXF[A,B]]*Y**C

B+16.427       (XBAR+(B[I,J+I,K]/3))

A LAN B        LNO A LOR B

LNO C + D      B * (LNO D)
```

In the following examples, I indicates an intermediate result (not a register)

LNO (A+B*(C-D*E-(-F+G)/3))

would be evaluated in a way <u>functionally</u> equivalent to:

$D * E \longrightarrow I_1$

$C - I_1 \longrightarrow I_2$

$-F \longrightarrow I_3$

$I_3 + G \longrightarrow I_4$

$I_4 / 3 \longrightarrow I_5$

$I_2 - I_5 \longrightarrow I_6$

$B * I_6 \longrightarrow I_7$

$A + I_7 \longrightarrow I_8$

$\text{LNO } I_8 \longrightarrow \text{Result}$

A**B/C+D*E*F-G is evaluated:

$\text{A**B} \longrightarrow I_1$

$I_1/C \longrightarrow I_2$

$D*E \longrightarrow I_3$

$I_3*F \longrightarrow I_4$

$I_2-G \longrightarrow I_5$

$I_4+I_5 \longrightarrow I_6$ evaluation completed

A**B/C(C+D)*(E*F-G) is evaluated:

$\text{A**B} \longrightarrow I_1$

$C+D \longrightarrow I_2$

$I_1/I_2 \longrightarrow I_3$

$E*F \longrightarrow I_4$

$I_4-G \longrightarrow I_5$

$I_3*I_5 \longrightarrow I_6$ evaluation completed

The following are examples of expressions with embedded parentheses:

A*(B+((C/D)-E)) is evaluated:

$C/D \longrightarrow I_1$

$I_1-E \longrightarrow I_2$

$B+I_2 \longrightarrow I_3$

$A*I_3 \longrightarrow I_4$ evaluation completed

(A*(SIN(X)+1.)-Z)/(C*(D-(E+F))) is evaluated:

SIN(X)	$\longrightarrow$	I_1	
I_1+1.	$\longrightarrow$	I_2	
A*I_2	$\longrightarrow$	I_3	
I_3-Z	$\longrightarrow$	I_4	
E+F	$\longrightarrow$	I_5	
D-I_5	$\longrightarrow$	I_6	
C*I_6	$\longrightarrow$	I_7	
I_4/I_7	$\longrightarrow$	I_8	evaluation completed

BOOLEAN EXPRESSIONS

The value of a Boolean expression is either TRUE or FALSE. Boolean expressions are used most often in IF statements.

RELATIONAL EXPRESSIONS

Relational expressions are a subset of Boolean expressions. A relational expression is used to compare the value of two arithmetic expressions or character operands.

Format:

a_1 op a_2

a_i are either arithmetic expressions or character operands.

op is a relational operator belonging to the list below.

Operator	Description	Symbol
EQ	Equal to	=
GR	Greater than	>
LS	Less than	<
GQ	Greater than or equal to	⩾
LQ	Less than or equal to	⩽
NQ	Not equal to	≠

A relation is TRUE if a_1 and a_2 satisfy the relation specified by the op; otherwise, it is FALSE. Boolean items are considered FALSE if the bits comprising that item are **all** zero; otherwise, they are considered TRUE. Boolean constants are integer 0 for FALSE and integer 1 for TRUE.

Relations are evaluated as illustrated in the relation p EQ q, which is equivalent to the question: Does p–q = 0? If the answer is yes, the relation is TRUE; otherwise FALSE. Relational expressions are converted internally to arithmetic expressions according to the rules of mixed-mode arithmetic. These expression are evaluated to produce the truth value of the corresponding relational expressions.

The order of dominance of the operand types within an expression is the order stated for mixed-mode arithmetic expressions.

In relational expressions +0 is considered equal to –0.

Examples:

```
A GR 16                R(I) GQ R(I-1)
R-Q(I) *Z LQ 3.14159   I NQ J(K)
B-C NQ D+E             I EQ (J(K))
```

LOGICAL EXPRESSIONS

Logical expressions are formed with Boolean operators and Boolean operands and have the values TRUE or FALSE.

The Boolean operators are listed below in order of precedence:

Boolean Operators	Description
NOT	Logical negation
AND	Logical conjunction
OR	Logical disjunction

Boolean operands include Boolean items, Boolean constants, Boolean functions, and relational expressions.

Evaluation of a Boolean expression is terminated as soon as evaluation of any part of the expression has determined the result. For example, if L_1 is FALSE in the logical expression L_1 AND L_2 AND L_3, then L_2 and L_3 are not evaluated, since the expression must perforce be FALSE as soon as any FALSE value is discovered.

The expression

A OR B AND NOT C

is evaluated:

NOT C $\longrightarrow B_1$

B AND $B_1 \longrightarrow B_2$

A OR $B_2 \longrightarrow B_3$

B_i are Boolean values; if B_3 is TRUE, the entire expression is TRUE.

If L_1, L_2, are logical expressions, the logical operators are defined as:

NOT L_1	FALSE only if L_1 is TRUE
L_1 AND L_2	TRUE only if L_1, L_2 are both TRUE
L_1 OR L_2	FALSE only if L_1, L_2 are both FALSE

Examples:

The algebraic expression $B-C \leqslant A \leqslant B+C$ may be written:

```
B-C LQ A AND A LQ B+C
```

An expression equivalent to the logical relationship $(P \rightarrow Q)$ may be written:

```
NOT (P AND (NOT Q))
```

A graphic representation of the operators is shown below:

ALPHA AND BETA

α	β	**Formula**
T	T	T
T	F	F
F	T	F
F	F	F

Figure 4-1. AND Boolean Operator

The OR Boolean operator indicates disjunction. A Boolean expression joined by OR is TRUE if either of its parts are TRUE, as shown in Figure 4-2.

ALPHA OR BETA

α	β	**Formula**
T	T	T
T	F	T
F	T	T
F	F	F

Figure 4-2. OR Boolean Operator

The NOT Boolean operator indicates negation. A Boolean expression with a leading NOT is TRUE only if the expression itself is FALSE. Figure 4-3 shows this graphically.

NOT ALPHA

α	**Formula**
T	F
F	T

Figure 4-3. NOT Boolean Operator

(ALPHA OR BETA) AND NOT (ALPHA AND BETA)

α	β	Formula
T	T	F
T	F	T
F	T	T
F	F	F

ALPHA OR BETA AND GAMMA

α	β	γ	Formula
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

(ALPHA OR BETA) AND GAMMA

α	β	γ	Formula
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

Figure 4-4. Combinations of Boolean Operators

OPERANDS AND MIXED-MODE OPERATIONS

BOOLEAN OPERANDS

Boolean operands and Boolean expressions differ in nature from arithmetic operands and expressions, and may not be involved with them in arithmetic expressions. No arithmetic operator will apply to any Boolean operand, and vice versa.

OPERAND TYPES – ARITHMETIC OPERANDS

SYMPL uses the following arithmetic operand types:

Real	Unsigned integer
Signed integer	Character

The hierarchy of operand types is the order listed above, with character operands being the lowest of the hierarchy and real operands being the highest. Character operands are not true arithmetic operands; they may be used only in relational expressions or with the operators LAN, LNO, LOR, LXR, LIM, and LQV. SYMPL does not access more than the first word of a character operand.

In general, the various arithmetic operators are applicable to operands of any type. Except as noted below, each individual operation is performed only on operands of the same type.

The compiler supplies conversion operations as appropriate, such that the common type of two operands affected by a single binary arithmetic operator is the higher of the two operand types involved. The result of such an operation is of the common type. Thus, the expression

```
I + R  (where I is integer and R is real)
```

is computed in floating point, after converting the value of I to floating point, whereas the expression

```
C  EQ  C<0,1>XYZ  (where C is a character variable)
```

is computed in character mode, with no conversion.

Similarly, the expression

```
(I+2) * R
```

is computed as follows:

1. Add 2 to I in integer mode

2. Convert the result of (1) to floating-point

3. Multiply the result of (2) by R, in floating-point.

Exceptions to the hierarchial conversion rule are the following:

ABS	The ABS intrinsic function operates only upon integers and real operands. The result of the operation on an integer is type unsigned integer; the result for any other argument type is the same type as the input argument.
Character operands	When an operand of type character is placed in combination with a noncharacter operand, typically it is converted to the type of that operational operand, as dictated by the hierarchy.
Exponentiation	Under certain circumstances, exponentiation may be performed in integer mode and will yield an integer result; this is true only for exponent operations with type integer base and type integer exponent. All other exponential operations are forced to the form: (real) ** (integer) and yields a real result.
Operators	The six operators (LNO, LAN, LQR, LXR, LIM, and LQV) operate without conversion upon operands of any type producing a result of type unsigned integer.

TYPE CONVERSION TRANSFER FUNCTIONS

The following information defines the techniques for conversion of operand type values for all operand type combinations.

CONVERSION OF CHARACTER OPERANDS

Character operands always exist in storage in an integral number of machine words padded on the right with blank characters. When combined for all arithmetic operations, character operands are converted to integer. The conversion to unsigned integer mode is identical to that of integer; and the conversion to real mode is equivalent to integer conversion followed by a conversion of the integer result to the desired final form.

When used in replacement statements, character operands will be either blank filled on the right or right truncated, as necessary.

When used in relations, character operands will be compared according to the display code value of their representation; and trailing blanks will not be significant.

When combined with logical operators (such as LAN), character operands will be truncated or blank filled, as necessary, to 10 characters.

CONVERSION OF INTEGER OPERANDS

Integer To Character Conversion

The rightmost byte of the integer is left justified in the receiving character field and the balance of that field (which may be long or nonexistent) is padded on the right with blanks.

Integer To Real Conversion

The integer is floated; if the integer is small enough to conserve precision, the conversion will result in a real number of the correct value.

Integer to Unsigned Integer Conversion

The SYMPL compiler does not perform a conversion from integer to unsigned integer.

Real Operand To Integer Conversion

Real operands are converted to integer by truncation, and converted to character if needed by first converting to integer and then to character. If the value represented by the real operand is larger than integer size, significance may be lost in this type of conversion.

Results of integer multiply or divide or the conversion from real to integer has the following limits:

$$-(2^{48}-1) \leqslant N \leqslant (2^{48}-1)$$

FUNCTION CALLS AND INTRINSIC FUNCTIONS

A function reference calls a subprogram, causes parameters to be passed to it, and represents a value whose type is specified by the declaration of the subprogram function.

The following functions are classified as intrinsic:

Name	Description
ABS	Absolute function
B or C	Bead function
LOC	Location function
P	Pointer function (see section 3)

ABS FUNCTION

The ABS function returns the absolute value of the argument. If the argument type is real, the returned value is also real; however, if the argument type is integer or unsigned, the returned value is unsigned integer. An argument of any other type merely returns an unmodified argument.

Format:

```
ABS(expr)
```

Example:

```
ABS (-17)
```

BEAD FUNCTION

If an item is viewed as a string of bits or bytes, accessing a segment of this string, essentially, is accessing beads of the string. The intrinsic bead functions allow reference to individual beads or group of beads.

The bead functions are the bit function (B) and the byte or character function (C).

Format:

$B<e_1,e_2>sb$ or $B<e_1>sb$

$C<e_1,e_2>sb$ or $C<e_1>sb$

e_1 and e_2 are arithmetic expressions specifying the first bead to be extracted and, in addition, the number of beads respectively. If e_2 is omitted, it is assumed to be 1.

sb is an item name or a subscripted array item specifying the source of the beads.

Beads are numbered from left to right, beginning with zero, and the size of a byte is six bits; thus, the bits of a word are numbered from 0 to 59 and the bytes from 0 to 9.

Both bits (B) and byte (C) functions can cross word boundaries when operations concern character operands. The bit function is limited to 10 characters, the byte function to 240 characters. No check is made for B or C functions extending beyond the operand. If the operand is not of type character, bead functions may not cross word boundaries.

		Maximum Values for $e_1 + e_2$	
Data Type		C	I,U,R,S
Function	$B < e_1, e_2 > sb$	60	60
	$C < e_1, e_2 > sb$	240	10

Bit function is considered to be type unsigned integer and the byte function type character independent of the type of sb. The appropriate type conversion applies to either bit or byte function results if used in an expression or on the right-hand side of a replacement statement.

Example 1:

```
ITEM BOAT C(20)  ;

C<0,3> BOAT = 'CAL' ;     ≡SETS THE FIRST THREE CHARACTERS OF
                           ITEM BOAT = 'CAL'≡
```

C<0,3> BOAT is of type character while B<0,18> BOAT is of type unsigned integer.

Example 2:

```
ITEM FLAGS;
FOR I=0 STEP 1 UNTIL 59 DO
     IF B<I,1> FLAGS EQ 1
     THEN GOTO L2;
≡LOOKING FOR MOST SIGNIFICANT 1-BIT≡
```

```
XYZ = B<30,30> L[I+1] ;

ARRAY TIME [0:7];

ITEM CARD C(0,0,10) ;

FOR J=0  STEP 1 UNTIL 7 DO

FOR I=0  STEP 1 UNTIL 9 DO

BEGIN

     GOTO SWITCH [C<I,1> CARD[J]] ;
     •
     ≡SWITCH ON CHARACTERS OF ITEM CARD≡
     •

END
```

LOC FUNCTION

The value of the intrinsic LOC function of a data structure is the address of the data structure during program execution.

Format of intrinsic LOC function:

    LOC(arg)

arg may be one of the following:

    item name

    subscripted array item

    procedure name

    function name

    switch name

    label name

    array name (with the optional subscriptor)

This is the only context in which an array name may appear with subscripts; the result is the address of the array entry with the subscript.

Examples:

```
P<BASE> = LOC (ARRAY) ;
P<BASE> = LOC (ARRAY[I]) ;
```

STATEMENTS 5

STATEMENT TYPES

Statements can be either simple or compound.

SIMPLE STATEMENTS

A single statement, such as a GOTO statement or an assignment statement, is called a simple statement. Thus, regardless of the complexity of the formula designating the right term of an assignment statement, it is a simple statement.

COMPOUND STATEMENTS

A series of simple statements may be grouped together and enclosed within the BEGIN . . . END brackets. Such groups are called compound statements.

A compound statement may encompass other compound statements. Each structure enclosed within matching BEGIN . . . END brackets is operationally a compound statement regardless of the nature of the inner structures.

Compound statements are often used as the controlled statements of the FOR and IF statements. A compound statement is not terminated with a semicolon.

STATEMENT CATEGORIES

Statements are grouped below into two categories, based upon the nature of the function they describe: Value Assignment statements (cause the assignment of value to items and array items) and Flow-of-Control statements (alter the normal sequential processing order for statements).

VALUE ASSIGNMENT STATEMENTS

The replacement statement and the exchange statement comprise this category.

Replacement Statement

The replacement statement is used to assign a value to an item.

Format:

```
          v=exp;
```

exp is an expression

v is one of the following:

- item name
- subscripted array item
- function name
- P-function reference
- bead function reference

A function name may be used as the left-hand side of a replacement statement only within the function of the same name.

The expression on the right is converted to the type of the left-hand side before making the assignment. See section 4 for details of conversion functions.

If the left-hand side is a bead function reference, only the specified beads will be replaced; the remainder of the item will remain unchanged.

Examples using the replacement statement:

```
ITEMA=TEMP;          ≡GIVE ITEMA THE VALUE OF TEMP≡

B=C NQ D AND E;      ≡COMPUTE THE VALUE OF C NQ D AND E
                       THEN GIVE THIS VALUE TO B ≡
```

Exchange Statements

Format:

v_1 = = v_2 ;

v_i are as in replacement statements except that function names are excluded.

Exchange statements cause the exchange of value, with appropriate type conversion, between the two named entities. The order of expansion is not guaranteed (either v_1 or v_2 may be stored first); however, the language does guarantee that expressions which must be evaluated to compute the address of v_1 or v_2 (subscript expressions, bead function component expressions) will be computed only once. Therefore, such expressions must be evaluated prior to the exchange and their values saved; the exchange process must refer to the expression values by referring to temporary variables. For example, the exchange statement A==B; is expanded to the form:

```
temp=A ;

A=B ;

B=temp ;
```

Also, the exchange statement

```
B<I,J>X[K] ==B<L,M>Y[N];
```

is expanded as follows:

```
1.   temp #1=I;
2.   temp #2=J;
3.   temp #3=K;
4.   temp #4=L;
5.   temp #5=M;
6.   temp #6=N;
7.   temp #7=B<(temp #1),(temp #2)>X[(temp #3)];
8.   B<(temp #1),(temp #2)>X[(temp #3)] =
                B<(temp #4),(temp #5)>Y[(temp #6)];
9.   B<(temp #4),(temp #5)>Y[(temp #6)] = (temp #7);
```

The temporary variables used for storage of component and subscript expressions are all of type integer; the temporary variable used for storage of the left side of the exchange statement assumes the type of the right side.

LABELS

A label is an identifier used to name a statement. It is declared in a label declaration but need not be declared prior to its use.

Format:

```
name:
```

Embedded blanks are not permitted between the name and the delimiter (:).

A label declaration may appear at any point in the program where it is legal for a statement to appear. If a label is immediately followed by a statement, it forms a name for that statement through which the compiled code generated by that statement can be accessed. If a label is not immediately followed by a statement, it forms an entry point for the next code to be generated. Generation of this code may be caused by the occurrence of a statement, or it may be code generated in response to some implicit program feature.

Examples of implicit code:

Test at the bottom of a FOR loop

Jumps around ELSE part of a conditional statement

Jumps around procedures, functions, and switch declarations

Format of labeled statement:

name:

or

name: statement

Since a labeled statement is itself a statement, two labels in sequence synonymous.

SWITCHES

A switch is a programmer named and defined entity which is a vector of label names and can be placed at the disposal of a jump.

ORDINARY SWITCHES

Format:

SWITCH name label_1, . . . ;

name is an identifier which names the switch.

label_i are label names.

In the simpler form of a switch, the label names which constitute the switch are ordered by their appearance in the switch list segment of the switch declaration; increasing integer values are associated with the points of the list, beginning with zero.

Examples of simple switch declarations:

```
SWITCH GONOGO PROCEED, QUIT;
SWITCH $WORD $ITEM,$ARRAY,$PROC,$FUNC;
SWITCH  AAA   LABEL1,  LABEL2,  LABEL3,  LABEL4,  LABEL5  ;
         ^
         └──── ≡DEFINES   A   SWITCH≡
GOTO    AAA[I] ;<───≡USES   A   SWITCH≡
 .           │
 .           │
         I=0 │
LABEL1:<─────┤    ≡GOTO   LABEL1≡
 .           │
 .       I=1 │
LABEL2:<─────┤    ≡GOTO   LABEL2≡
 .           │
 .       I=2 │
LABEL3:<─────┤    ≡GOTO   LABEL3≡
 .           │
 .       I=3 │
LABEL4:<─────┤    ≡GOTO   LABEL4≡
 .           │
 .       I=4 │
LABEL5:<─────┘    ≡GOTO   LABEL5≡
```

If it is known that the switch will never be accessed with certain values of the index, labels for these values may be omitted from the switch.

Example:

SWITCH LABLST L_1, ,L_3,L_4 ;

STATUS SWITCH

Format:

SWITCH name: status-name $label_1$: $status\text{-}value_1$, . . . ;

name is an identifier which names the switch.

status-name is the name of a status list.

$label_i$ is a label name.

$status\text{-}value_i$ is a status value chosen from the status list status-name.

In this form, the labeled names are ordered by the compiler according to their explicit association with status values from a specified status list.

The status switch is convenient when the argument is a status item; not all status values from the status list need be associated with labels in the switch declaration.

STATUS SWITCH DECLARATION

Example:

```
STATUS SIZE TINY,SMALL,MEDIUM,LARGE,ENORMOUS;

SWITCH FUN:SIZE LABEL1: TINY,LABEL5:ENORMOUS,
                LABEL2:SMALL,LABEL3:MEDIUM,
                LABEL4:LARGE;

GOTO FUN[SIZE'MEDIUM']; ≡GOTO LABEL3≡

     ≡USE WHEN ARGUMENT IS A STATUS ITEM≡
     •
     •
     •
LABEL3:    RETURN;
```

GENERAL

A switch is an ordered set of label names, each associated with an integer value. A GOTO statement specifying an argument is used to select the label from the switch associated with the value of the argument. If the value of the argument does not match any of the values associated with the switch list labels, the result is undefined; if the range checking option (option C) is selected on the SYMPL control statement, the job is terminated with an error message.

GOTO STATEMENT

The control statement GOTO directs the program to branch out of the normal, serial sequence of execution by transferring control to a statement designated by a label name or a switch name.

Format:

```
        GOTO label ;

           or

        GOTO switch[exp] ;
```

label is a label name

switch is a switch name

exp is an arithmetic expression

The first form, GOTO label, causes unconditional transfer of control to the point in the program associated with the given label name; the second form causes a label to be selected from the named switch according to the value of the argument expression:

Examples:

```
GOTO JAIL;

SWITCH GONOGO PROCEED, QUIT;

GOTO GONOGO [ 1];
```

The second GOTO statement is equivalent to:

```
GOTO QUIT;
```

CONDITIONALITY: IF STATEMENT

An IF statement causes a conditional transfer of control, depending on the value of a Boolean expression.

Format:

IF Bool-exp THEN statement_1

or

IF Bool-exp THEN statement_1 ELSE statement_2

Bool-exp is a Boolean expression

If the value of the Boolean expression is TRUE, statement_1 is executed. If it is FALSE the next sequential statement is executed; in the second format, statement_2 is executed if the value is FALSE.

When IF statements are nested, an ELSE clause always is associated with the inner most nested incomplete IF statement. (See examples 3 and 4.)

Example 1:

```
IF A EQ 1 THEN B = 2;

ELSE B = 0;

IF A EQ 1 THEN

IF B EQ 1 THEN

     GOTO L;

ELSE A = 7;

IF BOOL THEN

BEGIN
     BOOL = NOT BOOL ;
     GOTO L ;
END
```

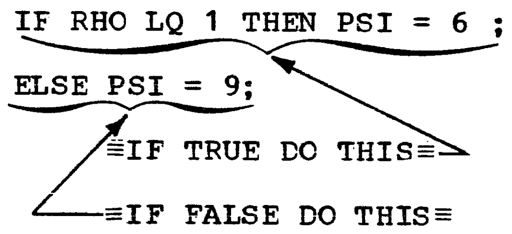
Example 2:

```
     IF AGE GQ 18 THEN

     GOTO VOTER;      ≡THIS STATEMENT IS EXECUTED IF
                      AGE IS GREATER THAN OR EQUAL
                      TO 18≡

     GOTO MINOR ;     ≡OTHERWISE THIS STATEMENT
                      IS EXECUTED≡

VOTER:
         •
         •
MINOR:
         •
         •
```

Example 3:

```
IF ≡1≡ A THEN
 IF ≡2≡ B THEN
  IF ≡3≡ C THEN
   ...ELSE ≡3≡...
  ...ELSE ≡2≡...
 ...ELSE ≡1≡...
```

Example 4:



```
IF RHO LQ 1 THEN PSI = 6 ;
ELSE PSI = 9;
    ≡IF TRUE DO THIS≡
    ≡IF FALSE DO THIS≡
```

```
IF A THEN
  IF B THEN
    IF C THEN D = 1;
    ELSE D = 2;
  ELSE D = 3;
ELSE D = 4;
```

LOOPING

The GOTO and the IF statements may be used to create program loops. Also, looping can be created through the FOR statement. In general, a program loop must perform five distinct steps:

Initialize	Set a counter and other program variables to initial values.
Test	Test whether counter has reached its terminal value.
Branch	Return to the execute step if the counter has not reached its terminal value; exit the loop if terminal value has been reached.
Execute	Perform necessary calculations for which the loop was constructed.
Modify	Change the counter or other variables by which loop iteration is controlled.

These steps are summarized in the following general flowchart:

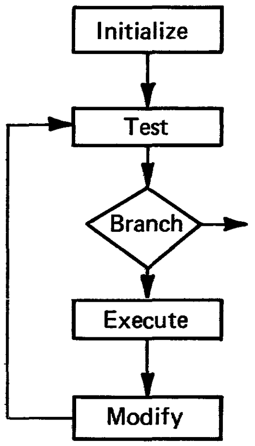


Figure 5-1. General Loop Flowchart

FOR STATEMENT

The FOR statement combines into one statement: the counter initialization, modification, testing, and subsequent branching. It also specifies a variable to be used as a counter. It can set the index to an initial value, declare the modification increment or decrement and set the terminal value. In addition to controlling the number of iterations performed, the index can be used also as an integer variable within the loop.

Format:

FOR i=x_1 DO statement

FOR i=x_1 STEP x_2 DO statement

FOR i=x_1 STEP x_2 UNTIL x_3 DO statement

FOR i=x_1 WHILE b_x DO statement

FOR i=x_1 STEP x_2 WHILE b_x DO statement

Item i, above, is called the induction variable, and it may be of any type except Boolean or character. Also, it will assume a value given as an initial value, and its subsequent value will control the execution of the repeated statement.

x_2 and x_3 are arithmetic expressions of unrestricted type; however, operations are carried out in the mode of the induction variable, with appropriate conversions.

b_x is a Boolean expression.

The intermediate language used to represent FOR statements is a straightforward expansion of code. The above cases expand to forms with the following SYMPL equivalents:

FOR I=X1 DO A=0;

is equivalent to

```
    I=X1;
L:  A=0;
    GOTO L;
```

FOR I=X1 STEP X2 DO A=0;

is equivalent to this sequence

```
    I=X1;
L:  A=0;
    I=I+X2;  GOTO L;
```

FOR I=X1 STEP X2 UNTIL X3 DO A=0;

is equivalent to sequence A, unless X2 has a minus sign prefix, in which case it is equivalent to sequence B.

A

```
    I=X1;
L:  IF I LQ X3 THEN BEGIN

    A=0;
    I=I+X2;  GOTO L;  END
```

B

```
    I=X1;
L:  IF I GQ X3 THEN BEGIN
    A=0;
    I=I+X2;  GOTO L;  END
```

FOR I=X1 WHILE BX DO A=0;

is equivalent to this sequence.

```
    I=X1;
L:  IF BX THEN BEGIN
    A=0;  GOTO L;  END
```

FOR I=X1 STEP X2 WHILE BX DO A=0;

is equivalent to this sequence.

```
    I=X1;
L:  IF BX THEN BEGIN
    A=0;
    I=I+X2;  GOTO L;  END
```

All tests are performed before execution of the controlled statement, which allows zero repetitions of the controlled statement.

It is possible to write a compound statement as the iterated statement of a FOR statement. The value of the induction variable is undefined after the loop is complete. However, if the iterated statement causes a jump out of the FOR statement, the current value of the induction variable at the time of the jump is preserved. Moreover, if the controlled statement is entered by a GOTO statement from outside the FOR statement, the value of the induction variable may be undefined.

Example:

```
          FOR A=1 STEP 2 UNTIL 99 DO

            IF VALUE[A+1] LS VALUE[A] THEN
              BEGIN
               .
               .
               .
     LOOP1:    VALUE[A+1] == VALUE[A] ;
               .
               .
               .
              END
```

If GOTO LOOP1; is executed from outside the BEGIN . . . END bracket, index A has not been initialized and the results of the exchange statement are undefined.

The statement controlled by a FOR statement may itself be a FOR statement, allowing for nesting:

Example:

```
     FOR A=1 STEP 1 UNTIL 10 DO

     FOR B=1 STEP 1 DO
         BEGIN
          .
          .
          .
          FOR C=1 STEP 2 UNTIL 60 DO
              BEGIN
               .
               .
               .
               FOR A=1 STEP 1 UNTIL 15 DO
                .
                .
                .
              END
         END
```

The above example, coded in error intentionally, illustrates an invalid use of the FOR clause; A is used as an induction variable for two loops at the same time.

TEST STATEMENT

In a FOR statement, the compiler automatically supplies the modification, testing, and branching steps of a loop. The TEST statement provides a means of branching to the implicit modify-test-branch steps as illustrated in the general flow-chart (figure 5-1).

Format:

TEST;

or

TEST name;

name is the name of an item used as an induction variable in a loop containing the TEST statement.

A TEST statement is meaningful only within the controlled statement of a FOR statement. When the TEST name statement is executed, control is transferred to the modify-test-branch for that induction variable; in this case, other index modify-test-branches could be skipped, and those induction variables would not be incremented for the next iteration. If name is omitted, control transfers to the modify-test-branch sequence of the innermost loop.

Examples:

```
FOR A=0 STEP 1 UNTIL 52 DO

    BEGIN
      •
      •
      •
    IF DEMAND[TODAY] GR DEMAND[TOMRW] THEN
    TEST;
      •
      •
      •

    END
```

If the conditional statement is TRUE, the TEST statement drops control to the increment step of the FOR loop, bypassing all coding between the TEST and END statements.

```
FOR A=0 STEP 1 UNTIL 100 DO
  BEGIN
      •
      •
      •
     FOR B=99 STEP -1 UNTIL 0 DO
     BEGIN
       IF INCOME GR 10000 OR CREDIT EQ S'GOOD' THEN
       TEST A;
       IF INCOME[B] GR OLDEST AND SEX[B] EQ S'FEMALE' THEN
       TEST B;
          •
          •
          •
     END

   END
```

If the conditions in the first IF statement are satisfied, control passes to the modify-test-branch for the outer loop, index A. If the first test statement had not specified A, control would have passed to the inner-most modify-test-branch, for B. If both conditions in the first IF statement are FALSE, execution bypasses the first TEST statement; and if the conditions of the second statement are satisfied, TEST B; statement is executed, passing control to the modify-test-branch for loop index B. Only when the above conditions are FALSE is the coding executed, that follows the TEST B; statement.

STOP STATEMENT

A STOP statement halts the program execution and returns control to the operating system. A STOP is generated automatically at the end of a main program.

Format:

```
STOP;
```

RETURN STATEMENT

The RETURN statement is meaningful only within a procedure or function. When a RETURN statement is executed in a procedure, control is returned to the calling routine.

Format:

```
RETURN;
```

PROCEDURE CALL STATEMENT

The procedure call statement is used to transfer control to a procedure, or closed subroutine, possibly passing data, and to set up return linkage to the calling routine.

Format:

name;

or

name(p_1, . . .);

name is a procedure name.

p_i are actual parameters

See section 7 for a more complete discussion of procedures.

COMPILER DIRECTIVES 6

CONTROL STATEMENT

The CONTROL statement directs the compiler to take immediate action. It may concern the compiler output by specifying a page eject or suppressing a source listing, or it may concern the generated code by causing a particular compiler option to be selected automatically. Additionally, the type of CONTROL statement known as a conditional-compilation directive, can cause the compiler to suppress the source code that immediately follows the directive.

The CONTROL statement is not executable at object time, although it may affect the contents of the object program. Also CONTROL statements may be introduced between an IF statement and its controlled statement.

The CONTROL statement may be written anywhere a statement can be written as well as in the following contexts:

Within a list of array item declarations enclosed by BEGIN. . . END brackets

Within a list of based arrays, enclosed by BEGIN . . . END brackets

Inside an external declarations list, enclosed by BEGIN . . . END brackets

Within a common declaration list, enclosed by BEGIN . . . END brackets

CONTROL statement format: (excluding conditional-compilation directive)

CONTROL control-word ;

The effect of the CONTROL statement is to perform the compiler action specified by the control-word as follows:

Control Word	**Function**
EJECT	Compiler skips to new page of listing output.
LIST	Compiler resumes normal listing of source.
NOLIST	Compiler suspends normal source listings.
OBJLST	Object code listing for this program is to be printed.
PACK	Turns on D option for this program.
PRESET	Turns on P option for this program.
FI ENDIF	(See page 6-2)

The OBJLST, PACK, and PRESET options apply to the entire program and the appropriate CONTROL statement should be placed at the beginning of the program.

CONDITIONAL COMPILATION

The format of a CONTROL statement specifying conditional compilation is as follows:

CONTROL condition-word $\underline{\text{constant}_1}$, $\underline{\text{constant}_2}$;

condition-word is defined below,

$\underline{\text{constant}_1}$ and $\underline{\text{constant}_2}$ are compile time constants.

This CONTROL statement causes optional compilation or suppression of source code, called conditional code. The code suppressed depends on the condition word as follows:

Condition-Word	**Compiles Conditional Code If**
IFEQ	$\text{constant}_1 = \text{constant}_2$
IFLS	$\text{constant}_1 < \text{constant}_2$
IFLQ	$\text{constant}_1 \leqslant \text{constant}_2$
IFGR	$\text{constant}_1 > \text{constant}_2$
IFGQ	$\text{constant}_1 \geqslant \text{constant}_2$
IFNQ	$\text{constant}_1 \neq \text{constant}_2$

Usually, one or both constants are specified by a DEF name or DEF parameter, thus parameterizing the conditional compilation. If constant_2 and its preceding comma are omitted, it is assumed to be integer zero. The constants may be integer, real, Boolean, or character, but both must be of the same type. No conversion of types takes place before comparison. Character constants may be compared only by the condition words IFEQ and IFNQ, and leading and trailing blank characters are considered significant.

Conditional code is bracketed between a conditional compilation directive and a CONTROL FI statement. The brackets may be nested, and source code is suppressed to a CONTROL FI that matches the conditional compilation directive. In any other situation CONTROL FI is ignored. CONTROL ENDIF is synonymous with CONTROL FI.

If conditional code is suppressed, syntax and semantic checks are not performed and DEF names are not expanded. Comment sequences and strings are not examined for the presence of start or end-of-conditional compilation directives. For this purpose, a semicolon (;) does not terminate a comment sequence.

Examples:

```
    DEF VERSION ≡3.4≡;
    DEF DBUG ≡1≡;
        .
        .
        .
    CONTROL IFNQ DBUG,1;            J=0; K=0; CONTROL FI;
    CONTROL IFGR DBUG,2;         IF J NQ 0 THEN K=K+1; ELSE K=K-1; CONTROL FI;
    CONTROL IFEQ VERSION, 2.0; RETURN; CONTROL FI;
        .
        .
        .
    CONTROL IFEQ VERSION, 3.4;
    CONTROL IFNQ DBUG; PRINT(CHECK);
    CONTROL FI;
    CONTROL FI;
        .
        .
        .
    CONTROL IFNE VERSION, 3.3;   FOR J=1 STEP 2 UNTIL N DO
        BEGIN . . .
            . . . END
    CONTROL FI;
    CONTROL IFNE VERSION, 3.4; ITEM X C(7); CONTROL FI;
    CONTROL IFEQ VERSION, 3.4; ITEM X C(8); CONTROL FI;
```

TERM STATEMENT

A TERM declaration signals the end of a compilation and must be the last statement of a program or subprogram.

Format of TERM statement:

```
    TERM
```

DEBUGGING CODE FACILITY

A programmer may want to include various source statements in his program which may be deleted easily from the compilation. The delimiters $BEGIN and $END allow the programmer to enter source statements which only will be compiled while the program is in the debug mode. They are syntactically identical to BEGIN and END; however, in certain circumstances, they can cause code to be deleted from the program.

Compound statements are of the form:

```
    BEGIN statements END
```

statements is composed of a sequence of zero or more statements (simple or compound); such a compound statement acts like a simple statement in every respect. Along with delimiting a compound statement, the special delimiters $BEGIN and $END bracket code which is to be optionally compiled.

When the compiler debug option is selected (E option on the compiler call statement), the delimiters $BEGIN and $END are identical in function to the standard delimiters BEGIN and END; the compiler option is selected when the compiled program should include the code enclosed within the special $BEGIN ... $END brackets. If the debug option is not selected, the coding between $BEGIN and $END is omitted from the compilation.

In normal mode, syntax is not checked for code appearing between $BEGIN and $END; code is not generated; declarations will have no effect on the code outside of $BEGIN and $END brackets.

The following restrictions must be observed:

The TERM statement must not appear in a $BEGIN . . . $END sequence and the $END must not result from a DEF expansion.

DEF DECLARATION

The unparameterized DEF declaration provides a source substitution capability by permitting an identifier to be defined as equivalent to a character string. When the identifier is used subsequently, it is replaced by the character string.

The parameterized DEF declaration provides a macro capability by allowing substitutable parameters to be associated with the definition of an identifier in a DEF declaration. The content of the replacing character string can be modified formally by varying the values of the substitutable parameters whenever the identifier is to be replaced by the character string.

UNPARAMETERIZED FORMAT

Format of unparameterized DEF declaration:

DEF name ≡character string≡ ;

name is an identifier called the DEF name.

The following restrictions apply to the DEF declaration:

No comment can be embedded between the name and the first quote mark of the character string (the quote being shown as the equivalence symbol ≡).

A quote within a DEF declaration is represented by two consecutive quotes.

Examples:

Legal	**Illegal**
DEF ON ≡1≡;	DEF FOX ≡FOXY≡]ONE≡;
DEF OFF ≡0≡;	DEF 1 A ≡DARK≡;
DEF BIT ≡ ≡;	DEF U ≡UNSIGNED≡;
DEF BOOL1 ≡A GR B AND B NQ O≡;	
DEF DEFINE ≡DEF≡;	
DEF REPL ≡A=B; ≡ ≡SET A=B≡ ≡ ≡;	

Whenever the DEF name is used at subsequent points in the program it will be replaced by the character string between the quote marks with the following exceptions:

Names that occur in defining contexts will not be replaced.

Descriptors and other single letter abbreviations (type descriptors B,C,I,R,S, and U; array layout specifiers P and S; the intrinsic functions B, C, and P; the constant prefixes O,S, and X; and the real number specifier E) will not be replaced.

Within a set of quotes ≡ . . . ≡

Within a set of primes ' . . . '

The DEF declaration may appear anywhere in the program that a normal SYMPL data declaration or imperative statement may appear and it is subject to the normal rules for declarations; the declaration must appear before the defined name is referenced and it has no effect outside the procedure within which it occurs.

A name defined by a DEF declaration is defined from that point for the remainder of the procedure. It may be redefined by the use of another DEF declaration for the same name at a subsequent point in the procedure, but it cannot be undefined. Thus, once a name has been given a definition for a particular program, there is no language structure whereby it may be returned to the usage it had before its first DEF declaration.

A defined name may be included in the character string defining another name. When defined names depend on one another for definition, the effect is the same regardless of the order in which the declarations are written; however, the position within the program is still important. A name can be defined only by the DEF declarations that precede its use in the program. Circular definitions are illegal.

Examples of nested DEFs:

Illegal nesting – circular definition:

```
DEF TWO     ≡BEGIN ONE END≡;

DEF ONE     ≡TWO≡;
```

Legal nesting:

```
DEF BOOL    ≡A AND B≡

DEF A       ≡C EQ 3≡

IF BOOL THEN X=1;
```

The above legal nesting example is equivalent to:

```
IF C EQ 3 AND B THEN X=1;
```

PARAMETERIZED FORMAT

Format of parameterized DEF declaration:

DEF name(<u>parm</u>$_1$, <u>parm</u>$_2$ . . . <u>parm</u>$_i$) "character string";

name is an identifier called the DEF name, and <u>parm</u>$_i$ are identifiers called DEF parameters.

The following restrictions apply to the parameterized DEF declaration:

No comment can be embedded between the DEF name and the left parenthesis of the parameter list.

No comment can be embedded between the right parenthesis of the parameter list and the first quote of the character string.

Within the character string, a quote is represented by two consecutive quotes; for example, the character string ≡A≡ ≡B≡ represents A≡B.

Examples:

```
Legal:      DEF M (X) ≡I=B+X≡;
            DEF BIT(I,J) ≡B<I>A[J]≡;
            DEF POPUP (STACK,TOP) ≡STACK[TOP]; TOP=TOP-1≡;

Illegal:    DEF M(X+Y) ≡I=B+X+Y≡;
            DEF BIT(17,K) ≡B<17>A[K]≡;
            DEF MIN ≡MACRO≡(X,Y,Z)≡ IF X LS Y THEN Z=X; ELSE Z=Y≡;
```

Parameterization of the DEF declaration occurs when the DEF parameter identifiers appear within a character string. They are assigned new values at each subsequent use of the DEF name. These values are character strings associated with the DEF parameters by an actual parameter list provided at every use of the DEF name. The DEF character string is modified by substituting the actual parameter values for the DEF parameter identifiers within the string.

A reference to a substitutable DEF parameter is recognized wherever a DEF parameter identifier occurs in the original DEF character string, unless it is enclosed in quotes or primes within the string. Apart from this restriction, a parameter identifier is substitutable whenever it appears, regardless of context. Care must be exercised, therefore, if a DEF parameter identifier is identical to any of the context dependent descriptors, functions, prefixes, or specifiers (B,C,E,I,O,P,R,S,U,X). Although each is an acceptable parameter identifier, its use will cause the corresponding context dependent significance to be unavailable within the DEF character string unless the final substituted value is itself acceptable in that context.

Examples: (substitutable parameters are underlined)

DEF BYTE(B,I,K) ≡$\underline{\text{B}}$<$\underline{\text{I}}$>A[$\underline{\text{K}}$]≡; ≡B may be substituted by C or B≡

DEF POINT(P,A) ≡Q=X[$\underline{\text{P}}$]; $\underline{\text{P}}$<$\underline{\text{A}}$>=Q≡; ≡likely to fail≡

PARAMETERIZED DEF EXPANSION

The replacement of the DEF name with the character string containing substitutable parameters is called the expansion. Expansion will not take place under the following circumstances:

If the DEF name occurs in a declarative context.

If the DEF name is also the name of a descriptor, function, prefix, or specifier and occurs in the correct context (B,C,E,I,O,P,R,S,U,X).

If the DEF name occurs within a set of quotes or a set of primes.

Additionally, the DEF name must be followed by a legitimate actual parameter list of the following form:

name (p_1, p_2, . . .)

name is a parameterized DEF name

p_i are actual parameters corresponding to the parameters in the DEF name declarations

No comment can be embedded between the DEF name and the left parenthesis of the actual parameter list

The actual parameters p_1, p_2, . . . are delimited initially by the left parenthesis and then by commas, or the terminating right parenthesis. Each actual parameter consists of the string formed by all characters between successive parameter delimiters. The resulting character strings, called parameter strings, will replace the corresponding DEF parameter identifiers wherever they are recognized in the DEF character string. If a parameter string contains incorrectly nested brackets or a semicolon, it may be bounded by quotes. The quotes are removed prior to substitution of the parameter. In such a parameter string, the quote is denoted by two quote symbols. A parameter starting with a quote is delimited by the matching " (quote).

Parameters that occur in a defining context will not be replaced. This situation is not detected until expansion time.

Commas and right parentheses (although they are parameter delimiters) may occur in parameter strings in non-delimiting circumstances. To be recognized as a parameter delimiter,a comma or a right parenthesis must be encountered at the outermost bracket level, otherwise it is considered to be part of the current parameter string. Within a parameter list,but not within quotes any of the characters [< (cause progression to an inner bracket level; and, conversely, any of the characters] >) cause a return to the previous bracket level. A parameter delimiter is not recognized between pairs of quotes or pairs of primes. Nor will characters between quotes or primes contribute to the bracket level. A semicolon may occur only between quotes or primes. There must be no net change in bracket level within an unquoted parameter string. Apart from these restrictions, the actual parameter list may contain any combination of characters. When a right parenthesis is encountered at the original bracket level (matching the left parenthesis in the above format), it delimits the current parameter string, and expansion of the DEF takes place.

The rules for forming parameter strings cause a parameterized DEF expansion to be syntactically similar to a procedure call or a function reference. Thus, all procedure or function actual parameter forms are acceptable DEF parameter strings. The converse is not true, however, since parameter strings are not constrained to be items or expressions, the sole restriction being that any bracketing characters used must be correctly paired.

If an actual parameter list contains more parameter strings than the number of DEF parameters specified in the DEF name declaration, a fatal diagnostic is issued and expansion does not occur. If fewer parameter strings are specified, expansion takes place with unspecified substitutable parameters replaced by the null parameter string, allowing the expansion of DEF names with a variable number of actual parameters.

Nested expansion of parameterized DEF names is permitted. However, recursive or circular expansion is prohibited.

Examples: Parameterized DEF name reference

```
M(I)

M(A+C[1,1]≡IGNORE≡)

BIT (B<3,2>,A[5])

BYTE(C,5,2**J)

CHECK(CALL(3,B),≡ERROR=37;GO TO FAIL≡);
```

Examples: Parameterized DEF expansion

The above reference to BYTE where:

```
DEF BYTE(B,J,K) ≡B<J>A[K]≡ ;
```

would expand as:

```
C<5>A[2**J]
```

The above reference to CHECK where:

```
DEF CHECK(X,ERROR) ≡IF BYTE(B,1,X) EQ 1 THEN GOTO OK; ERROR≡;
```

would expand as:

```
IF B<1>A[CALL(3,B)]EQ 1

THEN GOTO OK; ERROR=37; GOTO FAIL
```

However, the following definition of CHECK:

```
DEF CHECK(X,ERROR) ≡IF BYTE(B,1,≡ ≡X≡ ≡) EQ 1
                           THEN GOTO OK; ERROR≡;
```

Causes the above reference to expand as:

```
IF B<1>A[X] EQ 1

THEN GOTO OK; ERROR=37; GOTO FAIL;
```

PROGRAM STRUCTURE 7

PROCEDURES AND FUNCTIONS

SYMPL statements which perform a specific task may be combined for access as a single unit with procedure and function declarations; in addition, the declaration provides a means for transferring data to and from an accessed procedure or function. Procedure and function declarations may appear wherever any other declaration may appear.

PROCEDURE DECLARATION

A procedure declaration consists of a header followed by an optional series of declarations and a single statement, which may be compound. A procedure declaration is referred to as a procedure.

Format of procedure declaration header:

PROC name ($parm_1$,$parm_2$, . . .) ;

or

PROC name ;

$parm_i$ is an identifier called a formal parameter.

Examples:

```
PROC  CLEAR  (X,N)  ;

BEGIN
     ARRAY       X[99] ;

          ITEM  XX  R (0,0)  ;
          ITEM  N,  I        ;

     FOR  I=0  STEP 1 UNTIL N

     DO  XX[I] = 0.0 ;

END
```

```
PROC  REMQUO  (NUM,DEN,REM,QUO) ;
BEGIN
      ITEM  NUM,DEN,REM,QUO ;
      QUO = NUM/DEN ;
      REM = NUM-QUO*DEN ;
END
```

FUNCTION DECLARATION

A function declaration is similar to a procedure declaration and is referred to as a function; however, the function name, in addition to identifying the code, is associated with a specific value. A function declaration consists of a header followed by an optional series of declarations and a single statement, which may be compound.

Format of function declaration header:

FUNC name (parm$_1$,parm$_2$,. . .) type ;

or

FUNC name type ;

type is as defined for items and applies to function name value; if omitted, integer is assumed.

Examples:

```
FUNC    SIN(X)   R ;
      BEGIN
            ITEM  X  R ;
            .
            .
            .
            SIN = ... ;
            .
            .
            .
      END
```

```
FUNC     MOD(A,B)   U ;

     BEGIN

          ITEM  A  I,  B  I ;

          MOD = A-B*(A/B)  ;

     END
```

PROCEDURE AND FUNCTION USE

Procedure or function declaration statements are not executed at the point where they appear in the program. They must be explicitly called by a procedure call (see section 5) or function reference.

A function is automatically called when its name appears in an expression and the value computed by the function will be used when evaluating the expression. The value is associated with the function name by an assignment statement in which the function name is the left hand side and which appears within the function declaration.

The function name must not appear in any expression within the function declaration.

Procedure and function calls cannot be recursive; a procedure cannot call itself nor be called by any procedure which it calls.

Control is returned from a procedure or function to the calling routine when a RETURN statement (see section 5) is encountered or when the procedure declaration statement has been executed. Return from a procedure or function is normally to a point immediately following the call. It is possible to return to another point by a GOTO statement (see section 5) referencing a label in an outer procedure or function or formal label parameter.

Example:

```
PROC  XRAY  (X,Y,Z)  ;

     BEGIN

          ARRAY X [9:9] ;

               ITEM  XX  R ;
               ITEM  Y     ;

          GOTO Z;   ≡TAKE ALTERNATE EXIT TO FORMAL LABEL Z≡
          .
          .
          .
     END
```

FORMAL LABELS AND PROCEDURES

A use of a name will be associated with a prior declaration for the name if one exists, even if the declaration is at a more exterior level; this is true even for labels and procedures, for which forward definition is meaningful. Thus, a procedure declaration which uses labels or procedure names before declaring them is sensitive to declarations for other entities of the same name at outer levels. This situation may be avoided by designating such names prior to their usage in formal procedure or label declarations.

Example: Formal Label Declaration

```
PROC NAME ;

BEGIN

L1:
    PROC NAME1 ;

    BEGIN

    LABEL L1,L6 ;  ≡DECLARE FORMAL LABELS≡

    GOTO L1 ;      ≡AND YOU GO HERE, NOT HERE≡

L1: ... ;

    RETURN ;

    END

END

TERM
```

PARAMETERS

When a subprogram is called, a list of actual parameters is submitted by the call. In executing, a program operates on the actual parameters submitted by call. When a subprogram is declared, parameter handling is specified within the procedure body by reference to a set of dummy names, called formal parameters, listed with the subprogram declaration heading.

FORMAL PARAMETERS

The body of the subprogram uses formal parameters in much the same way that it uses other names; it must declare them before making use of them (except for labels and procedures) just as it must declare its own internal names. A declaration for a formal parameter, called a formal declaration, may occur anywhere before the name is used within the body of the subprogram.

SYMPL recognizes the following types of formal parameters: arrays, based arrays, functions, items, labels, and procedures. Each, with the exception of labels, requires formal declaration.

Formats for formal label, procedure, and function declarations:

LABEL $name_1$,$name_2$, . . . ;

FPRC name ;

FUNC name type ;

The format of a formal declaration for items, arrays, and based arrays is identical to the standard declarations, except preset information may not be included. A formal parameter that is an item may be referenced by address or by value.

To specify a call by value, the formal parameter name is enclosed in parentheses within the formal parameter list. A call by value gives more efficient access to the parameter but may not yield a result to its actual parameter. Only items may be called by value.

Example: Formal item declarations

```
PROC NAME (A,B,(C),(D));

≡A boolean, B character, C integer value, D real value≡

BEGIN ITEM A B;

   ITEM B C(5), C;
   ITEM D R;
   . . .;
      RETURN;

END
```

An item or array declaration within a subprogram is recognized as a formal declaration for a formal parameter only if its name coincides with the name of a formal parameter for the subprogram body where the declaration occurs. For example:

```
PROC  X(A) ;

BEGIN
     PROC  Y(B) ;
     BEGIN
          ITEM A ;
     END
     .
     .
     .
END
```

The declaration for A is not recognized as a formal declaration for parameter A of procedure X, rather it is treated as an item local to procedure Y.

A formal parameter may be declared to represent a based array; in this special case, the address of the pointer variable must be passed as a parameter; not the array location.

ACTUAL PARAMETERS

Actual parameters which are the arguments to a procedure or function when it is called are of the following types:

arithmetic expression	item name
array name	label name
Boolean expression	p function
function name	procedure name

In the parameter list position, item names and expressions should correspond to a formal parameter of type item; p function parameters should correspond to a formal based array; and the remaining parameter types should correspond to formal parameters of the same type.

Expressions are evaluated before subprogram execution and the address of a temporary location possessing the resulting value is passed as the parameter; other parameters are passed as addresses directly. A function name, without a parameter list, is not evaluated but is passed as an address to the called procedure or function. Note that a single array reference is considered an expression and evaluated.

Subprogram names and item names, without parameters, are normally passed as addresses. However, the logic differs somewhat if the names are parenthesized; they will be evaluated before the call and passed as temporary variables. The following example shows two calls on proc FUNNY, with the change in results caused by the use of parentheses.

Example: Formal and Actual Parameters

```
     ITEM J ;
     PROC FUNNY (FACE) ;
     BEGIN
          ITEM FACE ;
          ITEM A, B ;
          A=FACE ;
          J=3 ;
          B=FACE ;
EAR:      IF A EQ B THEN GOTO EAR ;
     END
          J=4 ;
          FUNNY(J) ;
```

Here a normal return is made to the calling program, whereas:

```
FUNNY ((J)) ;
```

causes the subprogram to loop endlessly at EAR.

Example: Formal and Actual Parameters

```
    A=1 ;
    B=2 ;
    NAME2(A,B) ;          ≡ACTUAL PARAMETERS≡
     .
     .
     .
                PROC  NAME2 (X,Y) ;  ≡FORMAL PARAMETERS≡
                     BEGIN
                     ITEM X, Y ;
                       .          ≡FORMAL ITEM DECLARATION≡
                       .          ≡LOCAL TO NAME2≡
                       .          ≡OR SCOPE NAME2≡
                     END
     .
     .
    END
```

SCOPE OF DECLARATION

Since the statement of a subprogram body may be compound, it may have embedded declarations, including subprogram declarations. Such declarations are nested within the main subprogram.

Names declared within the subprogram body are recognized only within that subprogram (and thus within other subprograms nested within it).

Thus, when nested subprograms contain declarations for the same name, the innermost declaration has precedence.

The scope of a declaration is the name of the subprogram within which it occurs.

EXTERNAL SUBPROGRAM

A procedure or function which is not nested within another subprogram is referred to as an external subprogram. An external subprogram must be terminated by the TERM directive and is compiled separately from other external subprograms. Also, the name of an external subprogram is automatically made available for reference by other subprograms through the XREF declaration.

MAIN PROGRAM

A main program consists of a program header followed by a series of declarations and statements. It must also be terminated by the TERM directive.

Format of main program header:

```
PRGM name ;
```

A main program is not called as is a subprogram since it provides the starting point for execution.

Only a TERM directive may precede a main program or external subprogram header and only a main program or external subprogram header may follow a TERM directive. There must be no intervening text.

ALTERNATE ENTRANCES: ENTRY DECLARATION

The ENTRY declaration within a subprogram body establishes an alternate entrance for the subprogram. It need not duplicate parameters associated with the subprogram name; however, the code associated with a given ENTRY should use only the parameters associated with that ENTRY, and values for parameters associated with other entries are undefined.

Format of ENTRY declaration:

ENTRY PROC name $(parm_1, parm_2, \ldots)$;

or

ENTRY FUNC name $(parm_1, parm_2, \ldots)$ type ;

parameter list and type are optional.

Example of an alternate entrance subprogram:

```
FUNC  F(X)  R ;

BEGIN

          ITEM X R ;
          F = X**X ;

     GOTO G1 ;

     ENTRY FUNC G(X)  I ;

          G=X**X ;

     G1:  X = X + 1.0 ;

END
```

```
BEGIN
     ITEM GENOPT ;
     DEF BUFMAX    ≡100≡     ;
     DEF BUFIX     ≡0≡    ;

     ARRAY CRBUF[BUFMAX] ;
          ITEM CRLIN (0, 42, 18) ;

     STATUS TYPLST DEF, SET, USE, SCP ;
     ITEM XLINK, XCARD, BUFIX, R, S ;  ≡DEFAULT TYPE IS
                                        INTEGER≡
     ITEM XTYPE S:TYPLST ;
     ROPTN ;         ≡FIRST EXECUTABLE INSTRUCTION≡
     XTYPE   =   S'DEF'    ;
     GOTO GO ;

          ENTRY PROC XRSET (XLINK, XCARD) ;  ≡ENTRY DEC≡
          ROPTN;
          XTYPE = S'SET' ;

          GOTO GO ;

               ENTRY PROC... ;

                    ENTRY PROC... ;
     GO:  CRLIN[BUFIX] = XCARD ;

     OUT:  RETURN ;

           PROC  ROPTN ;  ≡R-OPTION CHECKER≡
           BEGIN
               IF B<27,1>GENOPT EQ O THEN GOTO OUT ;
          RETURN;
END
```

INTERPROGRAM COMMUNICATION

Three SYMPL declarations allow communication between external subprograms: COMMON, XREF, and XDEF declarations.

These declarations may occur only at the outermost level of a compilation; names used in interprogram communication must be unique in the first seven characters and must not begin with the character $.

COMMON DECLARATION

SYMPL programs may declare variables to be assigned storage at load time using the COMMON declaration.

The COMMON declaration provides up to 61 blocks of storage that can be referenced by more than one subprogram, and the starting addresses for these blocks are indicated on the core map listing.

Format:

COMMON name ; data-declaration

or

COMMON name ; BEGIN data-declaration data-declaration . . . END

name is optional; if omitted, storage locations in blank common will be automatically assigned.

data declaration is either an item, array, or based array declaration.

Examples:

```
COMMON A ;  ITEM X ;

COMMON B ;  ARRAY TAB [50] ;  ITEM J ;

PRGM DEFCOMM ;

COMMON AREA ;

BEGIN

     BASED ARRAY AA; ITEM XX;
     ITEM A = 2955 ;
     ITEM B S:TEST = S'COND' ;
     ITEM C C(10) ;
     ITEM D I     ;
     ITEM E R     ;

     ARRAY [ 9]   ;
          BEGIN
               ITEM Q1STPNUM I (0,0,15) ;
               ITEM Q2NDPNUM I (0,15,15) ;
               ITEM BSTATNUM  I (0,30,30) ;
          END

     ITEM F ;
     ITEM G ;
     ITEM H ;
     ITEM I ;
     ITEM J ;


END
```

Presets may be included for named common blocks but not for blank common; however, the presets will be ignored unless the routine is compiled with the P option specified on the SYMPL compiler call statement or the CONTROL PRESET statement is used at the beginning of the routines.

Common declarations need not be identical in all routines referencing the common block; however, the routine with the longest block must be loaded first and relative locations for all items must be the same in all referencing routines. The following example shows how different declarations for the same items may be used to initialize a common block.

Example:

```
PROC ONE ;

BEGIN

     ITEM I ;

     COMMON BLOCK ;

     ARRAY ZERO [ 10 ] ;
     ITEM ZRO ;

     FOR I=0 STEP 1 UNTIL 10 DO

     ZRO[ I ] =0 ;

END

PRGM MAIN ;

     COMMON BLOCK ;

     BEGIN
          ITEM A1, A2, A3 :

          ARRAY BB [ 6] ;
          ITEM BLTM ;
     END

ONE ;
 •
 •
 •
```

EXTERNAL REFERENCE DECLARATION

The external reference declaration (XREF) is used to define items, arrays, based arrays, procedures, functions, labels, and switches which are actually part of another program. It is assumed that storage will be allocated elsewhere for names defined by XREF declarations.

External references which name the declared external entities are output with the object program, depending on the system's loader for the resolution. In this way, one compilation may define a set of procedures, for example, which may be referenced in other compilations by declaring them as XREF procedures.

Format of XREF declaration:

XREF xdec

or

XREF BEGIN xdec xdec . . . END

each xdec may be an item declaration without presets, array declaration without presets, based array declaration, or one of the following:

PROC name ;

FUNC name type ;

LABEL $name_1$,$name_2$, . . . ;

SWITCH $name_1$,$name_2$, . . . ,

type is optional.

Examples:

```
XREF ITEM XRAY R ;

XREF LABEL FAIL,SUCCESS ;

XREF ARRAY AUNUR [99] P(1) ;
     ITEM INTGR (0,0,60) ;

XREF BASED ARRAY AA ; ITEM XX ;

XREF PROC REMQUO ;

XREF BEGIN
     SWITCH JUMPVEC ;
     FUNC LINEUP R ;
     ITEM I, J, K, L ;
     ARRAY [0:9,0:9] S(5) ;
     BEGIN
       ITEM AA C(0,0,40) ;
       ITEM BB R(4,0,60) ;
     END

END
```

EXTERNAL DEFINITION DECLARATION

External definition (XDEF) declarations are defined and allocated storage in the current compilation and made available for reference, through the XREF declaration, in other compilations. The XDEF declaration is a complement of the XREF declaration for items and arrays. The compiler generates external definitions from XDEF declarations, allowing the system loader to tie together XDEF and XREF names.

Items, arrays, based arrays, procedures, functions, labels, and switches may be externally defined.

Format of XDEF declaration:

XDEF <u>xdec</u>

or

XDEF BEGIN <u>xdec</u> <u>xdec</u> . . . END

<u>xdec</u> may be an item, a switch, an array or based array declaration, or one of the following:

PROC name ;

FUNC name <u>type</u> ;

LABEL $name_1$,$name_2$, . . . ;

If Program A is compiled with:

```
XREF ITEM COUNT I ;
```

and Program B is compiled with:

```
XDEF ITEM COUNT I ;
```

references to the item COUNT within Program A actually will refer to the storage reserved for the item in Program B, assuming both programs A and B are available at load time.

XDEF declarations for procedure and function names may occur either before or after the declarations of the procedure or function.

Example:

```
XDEF  ITEM X ;

XDEF  BEGIN
      ITEM Y, Z ;
      ARRAY Q [99] ;
      FUNC ABS R ;
      PROC ZERO ;
      SWITCH JUMPVEC J1, J2, J3, J4 ;
      END        ≡XDEF DECLARATIONS≡
```

```
XDEF ARRAY LIBRARY  [0:9,-60:-50,1:1] ;
      BEGIN
         ITEM TITLE C(0,0,10) ;
         ITEM DEWDEC I ;
      END
```

Examples: MAIN PROGRAM AND SUBPROGRAM DECLARATIONS

```
PRGM  NAME ;  ≡MAIN PROGRAM HEAD≡
BEGIN
  .
  .
     PROC  NAME1 ;  ≡SUBPROGRAM DECLARATION≡
     BEGIN
          ITEM X ;  ≡X HAS NAME1 SCOPE≡
            .
            .
               PROC  NAME2 ;  ≡NESTED SUBPROGRAMS≡
               BEGIN
                 .
                 .
                 X=4 ;  ≡SAME X AS IN NAME1≡
                 RETURN ;
                 END
            .
            .
            .
          RETURN ;
     END
     PROC  NAME3 ;
     BEGIN
       .
       .
     RETURN ;
     END
  .
  .
STOP ;
END
TERM
```

```
PRGM  SORT100 ;

BASED ARRAY AA [99] ;
     ITEM X ;

XDEF PROC SORTER ;

ARRAY TOBESORTED [99] ;
     ITEM T ;

P<AA> = LOC(TOBESORTED) ;
     SORTER (P<AA>) ;

     PROC SORTER (SORT)

     BEGIN

          ARRAY SORT [99] ;
          ITEM VALUE ;
          ITEM FLAG I=0 ;

     L1:  FOR I=0 STEP 1 UNTIL 98 DO
          IF VALUE[I+1] GR VALUE[I] THEN

          BEGIN

               VALUE[I+1] == VALUE[I] ;
               FLAG = 1 ;

          END

          IF FLAG EQ 0 THEN

          RETURN  ;

               FLAG=0 ;

          GOTO L1 ;

     END

TERM
```

Example: EXTERNAL SUBPROGRAM (PROCEDURE)

```
PROC  NAME ;  ≡NAME PASSED TO THE LOADER≡

BEGIN
   .
   .
   .
      FUNC NAME1(A)  I ;  ≡FUNCTION SUBPROGRAM DECLARATION
                           AND TYPE OF THE RESULT≡


      BEGIN
      ITEM A ;
      NAME1=16*A**3 -4 ;
      RETURN ;
      END
   .
   .
   .
ITEM K ;
   .
   .
   .
B=14*NAME1(K)  ;    ≡CALL THE FUNCTION NAME1≡
   .
   .
   .
END

TERM
```

```
PROC  SUBROUT ;  ≡DEFINE SUBROUTINE≡

BEGIN

SWITCH  HIT  BALL1, BALL2, BALL3, BALL4 ;

XREF

     BEGIN


          PROC TEAM1 ;
          PROC TEAM2 ;
          PROC TEAM3 ;


     END

DEF CALL  ≡  ≠ ≠  ≡

IF XXX[I] EQ 0 THEN

     GOTO HIT[RUN] ;

     GOTO LEFTOUT ;

BALL1:          CALL TEAM1 ;
                RETURN     ;

BALL2:          CALL TEAM2 ;
                RETURN     ;

BALL3: BALL4:   CALL TEAM3 ;
                RETURN     ;


END

TERM
```

CALLING SEQUENCES

SYMPL uses standard calling sequences; a parameter list contains one word per parameter. The address of the parameter list is passed in register A1. Linkage is performed by executing an RJ instruction to the entry point. To provide compact object code, traceback information is not generated. The parameter list is not followed by a word of zeros, except when explicitly requested via the compiler call statement F option.

COMPILER CALL STATEMENT 8

The compiler call statement, which calls for the compilation of a SYMPL source program, consists of the characters SYMPL followed by an optional parameter list and terminated by a period or right parenthesis. The columns following the right parenthesis or period may be used for comments; they are transcribed to the DAYFILE. The format is:

```
SYMPL(p1,p2,p3,p4,p5,...)        comments

SYMPL. comments

SYMPL,p1,p2,p3,p4,p5,.... comments
```

PARAMETERS

The SYMPL compiler operates according to the options specified on the SYMPL compiler call statement; errors cause the compiler to abort. The following options may be specified:

SOURCE INPUT: I

If the source input parameter is omitted, source input is assumed to be on INPUT. Otherwise, this parameter must be provided:

I=lfn

lfn is the name of the logical file containing the source input. Source input parameters of the forms I=INPUT is equivalent to omitting the parameter. Specifying I alone is equivalent to I=COMPILE.

BINARY OUTPUT: B

If the binary output parameter is omitted, a relocatable binary file is written on a file named LGO. For any other output file, this parameter must be provided:

B=lfn

lfn is the logical file name on which the binary output is to be written. Binary output parameters of the form B=LGO, or B, are equivalent to omitting the parameter. B=0 suppresses generation of binary output.

OBJECT TIME LIBRARY SPECIFICATION: S

The LDSET table generated on the binary file may be changed with the S option:

S=0	Suppresses the LDSET table
S=AAA/BBB/CCC	LDSET table is produced with entries for libraries AAA, BBB, and CCC

Default is:

S=FORTRAN/SYSIO for CYBER 70/72-74 and CYBER 170

S=FORTRAN/SYMIO for CYBER 70/76

Example:

```
SYMPL (I,A,H,S=FTNLIB)
```

LIST: LXOR

If this parameter is omitted, a normal listing is provided on OUTPUT, including the source language and diagnostics. Other list options may be selected as follows:

a=lfn/l_1/l_2

a may be one or more of the following:

L	Normal listing, diagnostics follow source
X	Storage map, common block listing
O	Listing of generated object code; if not specified parameter l_1 and l_2 are not permitted
R	Cross-reference table, common block listing
lfn	Logical file name to receive the list output. All list output is suppressed if lfn=0. The listing appears on OUTPUT if lfn is omitted.
l_1	Line of user's code where object list is to start
l_2	Line of user's code where object list is to end

If not specified l_1 assumes the value 0 and l_2 the value $+\infty$, and the preceding / must be omitted.

When the default OUTPUT is used, lfn and the succeeding / may be omitted.

Example:

```
LO=100/200
```

is identical to

```
LO=OUTPUT/100/200
```

however, LO=/100/200 is in error.

Any combination (with no comma) of the above parameters provides the features indicated. LXOR=lfn specifies all options are to be listed on the given file and LO selects source and object listing on OUTPUT.

All listing is suppressed if L=0 is specified; L=1 results in a summary of system resources utilized.

TERMINATE COMPILATION: T

If this parameter is present, compilation will terminate following output of all source error messages. Since code will not be generated, options affecting code generation are ignored.

SINGLE STATEMENT SCHEDULING: W

Generally, this option requires more central processing time than normal multiple statement scheduling; but it preserves a closer correspondence between object code sequence and source code sequence, which is useful for code oriented debugging.

PRESETS IN COMMON: P

Presets for items declared in common normally are not placed in the object deck; but if they are to be present the P parameter is required.

COMPILE $BEGIN-$END CODE: E

Normally, object code is not compiled for SYMPL statements contained between the $BEGIN and $END syntactic brackets. If object code for these statements is to be generated, the E parameter is required.

PACKED SWITCHES: D

Normally, SYMPL switches are generated with one switch point per 60-bit word. When the D option is specified, however, two switch points are packed into a 60-bit word, reducing the code size but increasing overhead and execution time.

SWITCH RANGE CHECKING: C

Under normal conditions the SYMPL compiler does not provide a means for switch range checking; however, the C option causes a bounds check to be performed for each switch reference. If an out-of-bounds condition is detected, the compiler issues a diagnostic message and program execution is terminated.

Results are similar, if a null or unspecified switch point is selected along with the C option.

If the C option is omitted, a null switch point will produce either an endless loop condition or a mode error.

SUPPRESS DIAGNOSTIC: Y

This option suppresses the printing of the diagnostic message (semi ends comment) but not the corrective action associated with the message.

UNREFERENCED ITEMS IN CROSS REFERENCE: N

FORTRAN COMPATIBLE CALLING SEQUENCE: F

ABORT: A

The A option causes the compiler to abort at the end of the job step if it encounters errors.

COMPILE PROGRAM LIST: H

When this parameter is present, it overrides the CONTROL NOLIST command, causing all code to be listed.

SAMPLE DECK SETUPS FOR BATCH MODE

Compile producing listing on default OUTPUT file and binary output on default LGO file. Execute.

```
SAMPL1,T100,CM60000,P3.

SYMPL (LXOR)

LGO.

7/8/9       (END OF RECORD)

            (SYMPL SOURCE DECK)

6/7/8/9     (END OF FILE)
```

Compile producing listing on default OUTPUT file and binary output of a program and subprogram on default LGO file. Execute.

```
SAMPL2,T070,CM60000,P7.

SYMPL (XOR)

LGO.

7/8/9       (END OF RECORD)

            (SYMPL SOURCE DECK)

            (SYMPL SUBPROGRAM)

6/7/8/9     (END OF FILE)
```

Compile producing listing on default OUTPUT file; no execution, produce a binary card deck.

```
SAMPL3,T100,CM60000,P17.

SYMPL (LXOR,B=PUNCHB)

7/8/9       (END OF RECORD)

            (SYMPL SOURCE DECK)

6/7/8/9     (END OF FILE)
```

STANDARD CHARACTER SETS A

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation was specified when the operating system was installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE 1, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS 1, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of the standard character set table are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

STANDARD CHARACTER SETS

CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code	CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code
:†	:	00††	8-2	00	8-2	072	6	6	41	6	06	6	066
A	A	01	12-1	61	12-1	101	7	7	42	7	07	7	067
B	B	02	12-2	62	12-2	102	8	8	43	8	10	8	070
C	C	03	12-3	63	12-3	103	9	9	44	9	11	9	071
D	D	04	12-4	64	12-4	104	+	+	45	12	60	12-8-6	053
E	E	05	12-5	65	12-5	105	−	−	46	11	40	11	055
F	F	06	12-6	66	12-6	106	*	*	47	11-8-4	54	11-8-4	052
G	G	07	12-7	67	12-7	107	/	/	50	0-1	21	0-1	057
H	H	10	12-8	70	12-8	110	(	(	51	0-8-4	34	12-8-5	050
I	I	11	12-9	71	12-9	111	)	)	52	12-8-4	74	11-8-5	051
J	J	12	11-1	41	11-1	112	$	$	53	11-8-3	53	11-8-3	044
K	K	13	11-2	42	11-2	113	=	=	54	8-3	13	8-6	075
L	L	14	11-3	43	11-3	114	blank	blank	55	no punch	20	no punch	040
M	M	15	11-4	44	11-4	115	, (comma)	, (comma)	56	0-8-3	33	0-8-3	054
N	N	16	11-5	45	11-5	116	. (period)	. (period)	57	12-8-3	73	12-8-3	056
O	O	17	11-6	46	11-6	117	≡	#	60	0-8-6	36	8-3	043
P	P	20	11-7	47	11-7	120	[	[	61	8-7	17	12-8-2	133
Q	Q	21	11-8	50	11-8	121	]	]	62	0-8-2	32	11-8-2	135
R	R	22	11-9	51	11-9	122	%	%	63††	8-6	16	0-8-4	045
S	S	23	0-2	22	0-2	123	≠	" (quote)	64	8-4	14	8-7	042
T	T	24	0-3	23	0-3	124	→	_ (underline)	65	0-8-5	35	0-8-5	137
U	U	25	0-4	24	0-4	125	∨	!	66	11-0 or	52	12-8-7 or	041
V	V	26	0-5	25	0-5	126				11-8-2†††		11-0†††	
W	W	27	0-6	26	0-6	127	∧	&	67	0-8-7	37	12	046
X	X	30	0-7	27	0-7	130	↑	' (apostrophe)	70	11-8-5	55	8-5	047
Y	Y	31	0-8	30	0-8	131	↓	?	71	11-8-6	56	0-8-7	077
Z	Z	32	0-9	31	0-9	132	<	<	72	12-0 or	72	12-8-4 or	074
0	0	33	0	12	0	060				12-8-2†††		12-0†††	
1	1	34	1	01	1	061	>	>	73	11-8-7	57	0-8-6	076
2	2	35	2	02	2	062	≤	@	74	8-5	15	8-4	100
3	3	36	3	03	3	063	≥	\	75	12-8-5	75	0-8-2	134
4	4	37	4	04	4	064	¬	^(circumflex)	76	12-8-6	76	11-8-7	136
5	5	40	5	05	5	065	; (semicolon)	; (semicolon)	77	12-8-7	77	11-8-6	073

†Twelve or more zero bits at the end of a 60-bit word are interpreted as end-of-line mark rather than two colons. End-of-line mark is converted to external BCD 1632.

††In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations from ASCII/EBCDIC % yield a blank (55_8).

†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

SYMPL DIAGNOSTICS B

For errors that are detected during execution of the object program, diagnostic comments are printed in the source program listing produced by the compiler.

The compiler recognizes errors in SYMPL syntax; faulty programming logic is not recognized unless it produces a syntax error.

When the processor detects a source language error, it prints out the applicable diagnostic message number immediately preceding the line on which the error was detected. In addition, after the last source statement has been compiled, the compiler sums the total number of diagnostic messages encountered during compilation and prints out this message along with a detailed listing of each message number and the condition causing the error message.

COMPILER ABORT CONDITIONS

The compiler aborts on encountering a compiler call statement error. Also if field length is insufficient, the compiler aborts and issues the diagnostic SYMBOL TABLE OVERFLOW.

An attempt to compile an incorrect source program may cause an abort. When syntax and semantic errors of the program are corrected, the compiler will execute satisfactorily.

DAYFILE MESSAGES

–SYMPL–	xxxxx COMPILED	xxxxx is the PRGM/PROC name.
–SYMPL–	EMPTY INPUT FILE	The file specified by the I parameter on the compiler call statement is positioned at end of information.
–SYMPL–	COMPILER ABORT	See COMPILER ABORT CONDITIONS above.
–SYMPL–	BAD LOADER CALL	See COMPILER ABORT CONDITIONS above.
–SYMPL–	BAD EXP CALL TO FTN	See COMPILER ABORT CONDITIONS above.
–SYMPL–	PARAMETER nIN ERROR	Error occurred in nth parameter on the compiler call statement.

DIAGNOSTICS

The compiler diagnostic abbreviations, message numbers, and the condition causing the message are listed below:

DIAGNOSTIC ABBREVIATIONS

Abbreviation	Description
ID	Identifier
CHAR	Character
CHARS	Characters
DUP	Duplicate
DECL	Declaration
ILL	Illegal
HEX	Hexadecimal
CONS	Constant
PARENS	Parenthesis
STRG	String
SEMI	Semicolon
STMT	Statement
ERR	Error
SPECS	Specifications
PARAM	Parameter
EXPR	Expression
PROC	Procedure
PROG	Program
FUNC	Function
UNDECL	Undeclared
REF	Reference
REFS	References
EXPR	Expression
BOOL	Boolean
REPL	Replacement
/	Or
UNDECL	Undeclared
XDEF	External definition
XREF	External reference
1FXX	Conditional compilation computation word

COMPILER ERROR MESSAGES

Message Number	Condition Causing Message
001	LONG ID--FIRST 12 CHARS USED
002	DUP DECL--NEW ONE OVERRIDES
003	UNDECL ID DELETED
004	ILL OCTAL/HEX CONS
005	TERM MISSING
006	BAD STATUS CONS USE
007	BAD NESTING OF PARENS/BRACKETS
008	CRUD CHAR IN INPUT
009	CHAR STRG>240 BYTES--240 USED
010	ILL ARRAY ITEM ID USE DELETED
011	ILL SWITCH ID USE DELETED
012	ILL ARRAY ID USE DELETED
013	ILL STATUS LIST ID USE DELETED
014	ILL COMMON ID USE DELETED
015	SEMI MISSING AFTER ARRAY DECL
016	CRUD AT START OF STMT DELETED
017	ILL KEYWORD USE DELETED
018	ARRAY ITEM DECL LIST LACKS END
019	DUP DECL OVERRIDES
020	ITEM DECL ID ERR
021	DECL DISCARDED--SCAN RESUMES AT SEMI
022	ITEM DECL TYPE ERR--I ASSUMED
023	ILL ITEM LENGTH--1 BYTE USED
024	SIGNED PRESET ILL FOR THIS TYPE--IGNORED
025	SCAN RESUMES AT -BEGIN-
026	MISSING SEMI
027	ITEM PRESET ERR
028	SEMI ACCEPTED AS NULL STMT
029	BASED/XDEF/XREF ARRAYS NEED ID
030	ARRAY ITEM DECL SYNTAX ERR
031	ARRAY ITEM DECL TYPE ERR
032	BAD ARRAY BOUND VALUES--ASSUMED [0:0]
033	ARRAY BOUND SYNTAX ERR
034	ARRAY ITEM DECL PARTWORD SPECS ERR--DEFAULT TAKEN
035	ARRAY ITEM DECL 1ST BIT ALIGNMENT WRONG--0 USED
036	ILL ARRAY ITEM BOUNDARY--DEFAULT TAKEN
037	TOO MANY ARRAY ENTRIES
038	TOO MANY PRESET GROUPS
039	ARRAY PRESET SYNTAX ERR
040	COMMON/XDEF/XREF--AT OUTER SCOPE ONLY
041	BAD COMMON DECL IGNORED
042	BAD XREF/XDEF IGNORED
043	BAD BASED DECL IGNORED
044	XDEF/XREF LIST CRUD DELETED

045	SWITCH DECL SYNTAX ERR
046	COMMON LIST SCAN RESUMES AT -ARRAY-/-ITEM-
047	STATUS DECL SYNTAX ERR
048	-END- ENDS BAD COMMON LIST
049	DEF DECL SYNTAX ERR
050	BAD FORMAL PARAM DECL
051	PROGRAM BEGINS BADLY
052	PROG DECL LACKS ID
053	PROG DECL ERR--CRUD PRECEDES SEMI
054	XDEF/XREF LIST SCAN RESUMES AT LEGAL ENTRY
055	FORMAL LABEL DECL SYNTAX ERR
056	-END- ENDS BAD XDEF/XREF LIST
057	FORMAL PROC DECL SYNTAX ERR
058	FUNC DECL LACKS ID
059	FUNC DECL TYPE ERR--I ASSUMED
060	FUNC DECL LACKS SEMI
061	SCAN RESUMES AT SEMI
062	DUP FORMAL PARAM ID IN LIST
063	DUP PARAM ID--PRIOR DECL THIS SCOPE
064	PARAM LIST SYNTAX ERR
065	PROC DECL LACKS ID
066	PROC DECL SYNTAX ERR
067	UNDECL LABEL/PROC ID
068	FORMAL ID LACKS DECL
069	PARAM NOT USED IN THIS SCOPE
070	ILL DEF ID--NO EXPANSION
073	TOO MANY PARAM/ARRAY/ARRAY ITEM REFS
074	TOO MANY SUBSCRIPTS:SWITCH REF
075	NOT ENOUGH SUBSCRIPTS FOR ARRAY/ARRAY ITEM REFS
076	BAD SUBSCRIPT LIST
077	ILL LABEL/PROC ID USE DELETED
078	STATUS SWITCH DECL LACKS STATUS LIST ID
079	BAD LABEL USE IN STATUS SWITCH
080	STATUS SWITCH--VALUE TOO LARGE
081	STATUS SWITCH--DUP CONSTANT VALUES
082	STATUS SWITCH--MISSING CONSTANT
083	BEGIN/END MISMATCH. PROBABLE DISASTER
084	IF EXPR NOT BOOL
085	WHILE EXPR NOT BOOL
086	CRUD AFTER FINAL END IGNORED
087	-DEF- ID EXPANSION NEST TOO DEEP-ID DELETED
088	YOUR -DO- HAS BEEN FOUND
089	THE -THEN- HAS BEEN FOUND
090	MISSING -DO-
091	MISSING -THEN-
092	INITIAL VALUE EXPR ERR
093	-STEP- EXPR ERR
094	-UNTIL- EXPR ERR
095	-WHILE- EXPR ERR
096	BAD -GOTO- DELETED

097	BAD REPL STMT DELETED
098	PARTWORD VALUES AFTER FIRST 3 IGNORED
099	ITEM DISCARDED--SCAN RESUMES AT COMMA
100	HANGING -IF- CLAUSE
101	HANGING -FOR- CLAUSE
102	HANGING -ELSE-
103	EXTRA END--OMITTED BEGIN FOR SUBPROGRAM ASSUMED
104	ILL UNDECL PARAM USE DELETED
105	FOR STMT: INDUCTION ID ERR
106	-IF- EXPR ERR
107	DUP XDEF/XREF DECLS FOR ID
108	XDEF PROC/FUNC: NOT FULLY DECL
109	BAD FORMAL DECL
110	REDUNDANT FORMAL DECL
111	BAD PARAM LIST
112	BOOL ILL IN ARITH CONTEXT
113	COMMON LIST LACKS -END-
114	BASED LIST LACKS -END-
115	XDEF/XREF LIST LACKS -END-
116	COMMON LIST CRUD DELETED
117	BASED LIST CRUD DELETED
118	BASED LIST SCAN RESUMES WITH -ARRAY-
119	-END- ENDS BAD BASED LIST
120	0 LENGTH -DEF- STRING IGNORED
121	CHAR LENGTH OMITTED--1 ASSUMED
122	BAD ARRAY ENTRY SIZE
123	BRACKET NEST TOO DEEP
124	ILL EXPR TYPE THIS LEFT SIDE
125	BAD BEAD FUNC
126	EXPR OP CONCATENATION ERR
127	LONG CHAR STRG--240 BYTES USED
128	BAD -LOC- FUNC
129	BAD -ABS- FUNC
130	BAD INDUCTION ID TYPE
131	NON INDUCTION ID IN -TEST-
132	-TEST- ILL OUTSIDE LOOP
133	SCAN RESUMES AT -BEGIN-/-ITEM-/SEMI
134	READ FUNC NEEDS ID
135	DUP STATUS ID
136	SEMI ENDS COMMENT
137	CONTROL STMT SYNTAX ERR
138	CHAR NOT D/F IN REAL OR DOUBLE CONSTANT
139	FORMAL PARAM PRESET ILL
140	XREF PRESET ILL
141	BLANK COMMON PRESET ILL
142	BASED ARRAY ITEM PRESET ILL
143	BAD P-FUNC

144	CHARACTER ITEM >240 BYTES - 240 USED
145	NO SUBSCRIPT FOR ARRAY ITEM - 0 USED
146	CIRCULAR DEF NAME EXPANSION - EXPANSION IGNORED
147	NO MAIN PROC FOR ENTRY PROC
148	ILLEGAL CHAR IN MACRO DEF
149	ILLEGAL IFXX COMPARE
150	TOO MANY DEF PARAMS
151	ILLEGAL CONDIT DIRECTIVE IGNORED
152	ILLEGAL VALUE PARAM-LABEL
153	ILLEGAL VALUE PARAM-ARRAY
154	ILLEGAL VALUE PARAM-PROC
155	COMMON BASED ARRAY DECL ERROR
156	LABEL DECL ERROR
157	XREF SWITCH ERROR
158	UNMATCHED IFXX
159	DEF PARAM ERROR
160	([OR < NESTING TOO DEEP
161	([OR < NEST MISMATCH
162	PARAMETER TOO LONG
163	PARAMETER COUNT ERROR
164	RECOVERY AT ;
708	ZERO-DIVIDE ATTEMPT

GLOSSARY C

The following glossary includes short descriptions of use for each SYMPL word, each special use of single letters, and each mark.

ABS	Intrinsic function. Returns the absolute value of the argument.
AND	Boolean operator. When used in x and y, yields TRUE only if x and y are both TRUE; otherwise, yields FALSE.

AND	0	1
0	0	0
1	0	1

ARRAY	Declares dimensioned entities.
B	Two uses: In item declarations, denotes Boolean type which is represented by a bit string whose values are interpreted: One = TRUE and zero = FALSE. Intrinsic function – Accesses bits of a variable
BASED	Declares arrays that have an explicit pointer variable but no compiler assigned storage.
BEGIN	Left bracket for the BEGIN . . . END pair. Delimits a compound statement.
C	Two uses: In item declarations, denotes character type which is represented by a bit string whose values are interpreted: one = TRUE and zero = FALSE. Intrinsic function – Accesses bytes of a variable.
COMMON	Delimits declarations for variables assigned storage by the loader, not the compiler. Common blocks of the same name share the same physical storage at execution time.
CONTROL	Introduces a compiler directive.
CONTROL IFxx	EQ, NQ, . . . Conditional compilation directives.

DEF	Declares a macro.
DO	Separates the FOR clause from the simple or compound statement executed under the control of the induction variable as declared for the FOR statement.
ELSE	Marks the statement to be executed on the FALSE evaluation of the Boolean expression in the IF statement.
END	Right bracket of BEGIN . . . END pair.
ENTRY	Declares alternate entrances to procedures and functions.
EQ	Relational operator. Denotes relationships of equality as in a EQ b, which is TRUE only if a a is algebraically equal to b.
FALSE	Boolean constant having the integer value 0.
FOR	Delimits start of FOR statements.
FPRC	Formal procedure declarator.
FUNC	Declares a function subprogram; a closed routine that returns a value to the expression of which the function call is part. FUNC as the first statement of a compilation declares the function to be stand-alone, which renders the function name available to the system loader. Also formal function declaration.
GOTO	Unconditional branch operator. Also used to access switches.
GQ	Relational operator. Denotes greater or equal relationships; a GQ b is TRUE if a is algebraically greater than or equal to b.
GR	Relational operator. Denotes greater relationships. Thus, a GR b is TRUE only if a is algebraically greater than b.
I	Denotes type integer in item declarations.
IF	Delimits start of conditional statement.
ITEM	Declares SYMPL variables. Item types include signed integer (I), real (R), status (S), Boolean (B), character (C), and unsigned integer (U).
LABEL	Formal label declarator.
LAN	Arithmetic operator. Bit-by-bit "and" of integer operands (see AND).

LIM Airthmetic operator. Bit-by-bit "implies" of integer operands.

A LIM B	A=0	A=1
B=0	1	0
B=1	1	1

LNO Unary arithmetic operator. Bit-by-bit complement of integer operands.

LOC Intrinsic function. Returns the location of a label, array, array entry, switch, procedure, function, or variable.

LOR Arithmetic operator. Bit-by-bit "or" of integer operands (see OR).

LQ Relational operator. Denotes relationships of less or equal. Thus, a LQ b is TRUE if a is algebraically less than or equal to b.

LQV Arithmetic operator. Bit-by-bit equivalence for integer operands.

LQV	0	1
0	1	0
1	0	1

LS Relational operator. Denotes relationship of less than. Thus, a LS b is TRUE only if a is algebraically less than b.

LXR Arithmetic operator. Bit-by-bit exclusive OR of operands.

LXR	0	1
0	0	1
1	1	0

NOT Boolean operator. Unary operator that effects the complement of the single Boolean operand. Thus, NOT a is TRUE only if a is FALSE; NOT (a AND b) is TRUE if either a or b is FALSE.

NQ Relational operator. Denotes relationships of not equal. Thus, a NQ b is TRUE only if a is not algebraically unequal to b.

O Octal constant prefix. Provides octal constants.

OR	Boolean operator. Binary inclusive or relationships. Thus a or b is evaluated TRUE if either a or b is TRUE; otherwise, it is FALSE.

OR	0	1
0	0	1
1	1	1

P	Two uses: In array declarations, denotes parallel array structure, meaning successive instances of the same item are in contiguous storage. (See serial, S.) Intrinsic function. Allows reference to the pointer variable.
PRGM	First word of a main program compilation. It declares the compiled output to be a program rather than a subprogram.
PROC	Declares a procedure subprogram. Parameters may be passed to and from such subprograms. PROC as the first word in a compilation creates the procedure as a stand-alone subprogram, and the procedure name is made available to the system loader.
R	Used in item declarations to denote type real (floating point).
RETURN	In a subprogram, causes exit to be made to the routine calling the subprogram.
S	Two uses: Status – In item declarations, denotes status type. Serial – In array declarations, denotes serial array structure, which means that different items of the same entry are in contiguous storage.
STATUS	Delimits a status declaration.
STEP	Separates the initial value expression of a FOR clause from the increment expression.
STOP	Returns control from a SYMPL program to the operating system.
SWITCH	Declares a vector of switch points with which the compiler associates indexes.
TERM	Compiler termination delimiter.
TEST	Used within the range of a FOR statement to effect a jump to the increment and test of the induction variable.

THEN	Denotes the statement (simple or compound) to be executed on the TRUE evaluation of the Boolean expression in an IF statement.
TRUE	Boolean constant having the integer value 1.
U	Denotes type unsigned integer in item declaration.
UNTIL	In FOR statements, separates the STEP (increment) expression from the final value of the induction variable.
WHILE	In FOR statements, delimits a Boolean expression for which TRUE evaluation causes FOR loop iteration, and FALSE evaluation terminates the loop.
X	Hexadecimal constant prefix.
XDEF	Delimits variables whose names and locations are to be made available to the system loader. Other separately compiled programs and subprograms may refer to such SDEF variables through XREF declarations.
XREF	Delimits variables and subprograms whose locations are to be supplied by the system loader through XDEF variables in other compilations.
$BEGIN	Brackets code to be compiled on option.
$END	Brackets code to be compiled on option.
+	Arithmetic operator. Add.
-	Arithmetic operator. Binary subtraction, or unary negation.
*	Arithmetic operator. Multiply. x*y reads: x multiplied by y.
/	Arithmetic operator. Divide. x/y reads: x divided by y.
**	Arithmetic operator. Exponentiation. x**(y+2) reads: x raised to the y+2 power.
=	Assignment operator. Denotes replacement. x=y reads: replace x with the current value of y.
= =	Assignment operator. Denotes exchange. x= = y reads: exchange the values of x and y.
,	Separates expressions, list elements, etc.
.	Decimal point in real constants.
:	Delimits labels and separates bound pairs in array dimensions. Other miscellaneous uses.

;	Terminates simple statements and declarations.
(Blank)	Used for syntactic readability.
( )	Parentheses (left and right). Used to bracket arguments to functions, procedures and parameterized DEF. Also used to group expressions and to denote call by value from parameter. Used elsewhere for syntactic readability.
[ ]	Brackets (left and right). Used to bracket subscripts.
′	Prime (left and right). Brackets character constants. Also encloses octal, hexadecimal and status constants.
“ ”	Quote (left and right). Brackets comments and right sides of definitions.
< >	Delimiters (left and right). Used to bracket arguments for some intrinsic functions (P, B, C).
″ (or ≡)	Quote represented throughout manual by equivalence symbol ≡. Bracket comments and right sides of definitions.

METALANGUAGE D

METALANGUAGE DESCRIPTION

The mechanics for defining the syntactic forms of SYMPL are accomplished through an elementary descriptive language, capable of defining any phrase-structured language.

SYMPL is described in a metalanguage by a set of statements called productions, each of which describes one form belonging to SYMPL. The forms of a language are its syntactic entities, such as the sentence or adverbial phrase (from English), or arithmetic expressions (from FORTRAN, for example).

Every form of SYMPL is described by one metalinguistic production.

Format of a production is as follows:

$\underline{\text{form name}}$:= context ⅃ form definition L context

$\underline{\text{form name}}$	Underscored name of the form defined by this production. In the metalanguage every underscored sequence is a form name.
:=	Production symbol, which may be read: has the form.
form definition	Structure of the form defined by this production (whose name is given as the form name of the production). The definition of a form specifies the set of character sequences (utterances) that it represents; form definitions specify a sequence of the following entities: Characters of the SYMPL character set, which represent themselves. Names of SYMPL forms, which represent sequences of characters of the SYMPL character set, as specified by the productions which describe the form names.

Sets of entities like the above, from which any one may be chosen. Such a set is enclosed within braces to indicate alternatives. The use of such alternative sets may be recursive defined; thus the form definition

$$\underline{X} \left\{ \begin{matrix} P & \left\{ \begin{matrix} \underline{R} \\ S \end{matrix} \right\} \\ \underline{Q} & \end{matrix} \right\}$$

is equivalent to a choice of one of the following alternative sequences:

$$\begin{matrix} \underline{X} & P & \underline{R} \\ \underline{X} & P & S \\ \underline{X} & \underline{Q} & \end{matrix}$$

The null form ϕ represents zero characters of SYMPL. Typically, ϕ is used as one member of an alternative set if no member of the set must be chosen.

context⌋ and ⌊context	Optional constraints upon applicability of the production. If a production contains either or both context sequences, the specified form name only represents the sequence of SYMPL characters defined by form definition when it occurs in the given context. A context sequence is formed similarly to a form definition sequence.

Thus, the production pair

$$\underline{X} \quad := \quad \left\{ \begin{matrix} & A & \\ \underline{X} & & A \end{matrix} \right\}$$

$$\underline{Y} \quad := \quad B \rfloor \quad \underline{X} \quad \lfloor B$$

describes sequences of the character A as the form name Y only when they are delimited by occurrences of the character B.

To summarize: seven symbols are peculiar to the metalanguage:

Underscore line	__________
Production symbol	:=
Null symbol	ϕ

Braces	{	and	}
Context delimiters	⌋	and	⌊

All other printed characters in metalinguistic productions are either form names (underscored) or self-representative members of the SYMPL character set.

BASIC NOTATION AND ELEMENTS

CHARACTER SET

SYMPL programs are composed of 55 characters, as follows:

LETTERS

$$\underline{\text{letter}} \ := \left\{ \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \\ \text{E} \\ \text{F} \\ \text{G} \\ \text{H} \\ \text{I} \\ \text{J} \\ \text{K} \\ \text{L} \\ \text{M} \\ \text{N} \\ \text{O} \\ \text{P} \\ \text{Q} \\ \text{R} \\ \text{S} \\ \text{T} \\ \text{U} \\ \text{V} \\ \text{W} \\ \text{X} \\ \text{Y} \\ \text{Z} \\ \$ \end{array} \right\}$$

DIGITS

$$\underline{\text{digit}} \quad := \quad \left\{\begin{matrix}0\\1\\2\\3\\4\\5\\6\\7\\8\\9\end{matrix}\right\}$$

MARKS

$$\underline{\text{mark}} \quad := \quad \left\{\begin{matrix}*\\/\\+\\-\\(\\)\\[\\]\\<\\>\\\text{"}\\\text{'}\\=\\.\\,\\;\\:\\\text{ƀ}\end{matrix}\right\}$$

ƀ represents a blank space. " is represented throughout manual as the equivalence symbol ≡.

BLANK SPACES AND COMMENTS

$$\underline{\text{space}} \quad := \quad \left\{\begin{matrix}\text{ƀ}\\\underline{\text{comment}}\end{matrix}\right\}$$

$$\underline{\Delta} \quad := \quad \left\{\begin{matrix}\underline{\text{space}}\\\underline{\Delta}\ \underline{\text{space}}\end{matrix}\right\}$$

$$\underline{\text{v}} \quad := \quad \left\{ \begin{array}{c} \underline{\wedge} \\ \phi \end{array} \right\}$$

$$\underline{\text{comment}} \quad := \quad \text{"} \; \underline{\text{comment string}} \; \text{"}$$

$$\underline{\text{comment string}} \quad := \quad \left\{ \begin{array}{l} \psi \\ \underline{\text{comment string}} \;\; \psi \\ \phi \end{array} \right\}$$

ψ represents any key punch character except semicolon (;) and quote ("), either of which will terminate a comment.

The forms $\underline{\wedge}$ and $\underline{\vee}$ are used throughout the metalinguistic description to represent one or more blanks and zero or more blanks, respectively.

IDENTIFIERS

$$\underline{\text{ident}} \quad := \quad \underline{\text{mark}} \;\; \rfloor \;\; \underline{\text{ident part}} \;\; \lfloor \;\; \underline{\text{mark}}$$

$$\underline{\text{ident part}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{letter}} \\ \underline{\text{ident part}} \qquad \left\{ \begin{array}{c} \underline{\text{letter}} \\ \underline{\text{digit}} \end{array} \right\} \end{array} \right\}$$

RESERVED SYMBOLS

The 52 SYMPL words are represented as follows:

abs	:=	mark	⌋	ABS	⌊	mark
and	:=	mark	⌋	AND	⌊	mark
array	:=	mark	⌋	ARRAY	⌊	mark
based	:=	mark	⌋	BASED	⌊	mark
begin	:=	mark	⌋	BEGIN	⌊	mark
common	:=	mark	⌋	COMMON	⌊	mark
control	:=	mark	⌋	CONTROL	⌊	mark
def	:=	mark	⌋	DEF	⌊	mark
do	:=	mark	⌋	DO	⌊	mark

else	:=	mark	⌋	ELSE	⌊	mark
end	:=	mark	⌋	END	⌊	mark
entry	:=	mark	⌋	ENTRY	⌊	mark
eq	:=	mark	⌋	EQ	⌊	mark
false	:=	mark	⌋	FALSE	⌊	mark
for	:=	mark	⌋	FOR	⌊	mark
fprc	:=	mark	⌋	FPRC	⌊	mark
func	:=	mark	⌋	FUNC	⌊	mark
goto	:=	mark	⌋	GOTO	⌊	mark
gq	:=	mark	⌋	GQ	⌊	mark
gr	:=	mark	⌋	GR	⌊	mark
if	:=	mark	⌋	IF	⌊	mark
item	:=	mark	⌋	ITEM	⌊	mark
label	:=	mark	⌋	LABEL	⌊	mark
lan	:=	mark	⌋	LAN	⌊	mark
lim	:=	mark	⌋	LIM	⌊	mark
loc	:=	mark	⌋	LOC	⌊	mark
lor	:=	mark	⌋	LOR	⌊	mark
lno	:=	mark	⌋	LNO	⌊	mark
lq	:=	mark	⌋	LQ	⌊	mark
lqv	:=	mark	⌋	LQV	⌊	mark
ls	:=	mark	⌋	LS	⌊	mark
lxr	:=	mark	⌋	LXR	⌊	mark
not	:=	mark	⌋	NOT	⌊	mark
nq	:=	mark	⌋	NQ	⌊	mark
or	:=	mark	⌋	OR	⌊	mark
prgm	:=	mark	⌋	PRGM	⌊	mark
proc	:=	mark	⌋	PROC	⌊	mark
return	:=	mark	⌋	RETURN	⌊	mark
status	:=	mark	⌋	STATUS	⌊	mark

step	:=	mark	⌋	STEP	⌊	mark
stop	:=	mark	⌋	STOP	⌊	mark
switch	:=	mark	⌋	SWITCH	⌊	mark
term	:=	mark	⌋	TERM	⌊	mark
test	:=	mark	⌋	TEST	⌊	mark
then	:=	mark	⌋	THEN	⌊	mark
true	:=	mark	⌋	TRUE	⌊	mark
until	:=	mark	⌋	UNTIL	⌊	mark
while	:=	mark	⌋	WHILE	⌊	mark
xdef	:=	mark	⌋	XDEF	⌊	mark
xref	:=	mark	⌋	XREF	⌊	mark
spbegin	:=	mark	⌋	$BEGIN	⌊	mark
spend	:=	mark	⌋	$END	⌊	mark

The action of $BEGIN and $END depends on the presence of option E on the SYMPL control statement.

SPECIAL IDENTIFIERS

array item name	:=	ident
array name	:=	ident
based array name	:=	ident
common name	:=	ident
def name	:=	ident
formal array name	:=	ident
formal based name	:=	ident
formal func name	:=	ident
formal item name	:=	ident
formal proc name	:=	ident
func name	:=	ident
item name	:=	ident
label name	:=	ident

proc name	:=	ident
program name	:=	ident
status list name	:=	ident
switch name	:=	ident

DEF DECLARATIONS

DEF SPECIFICATION

def head	:=	def∧ident
defmac head	:=	def head opt space (∨ def params ∨)
def dec	:=	def head opt space "non quote string" ∨ :
defmac dec	:=	defmac head opt space "non quote string"
opt space	:=	$\left\{\begin{array}{l}\phi \\ \text{opt space } ƀ\end{array}\right\}$
non quote string	:=	$\left\{\begin{array}{l}\psi \\ \text{non quote string } \psi\end{array}\right\}$
def params	:=	$\left\{\begin{array}{l}\text{ident} \\ \text{def params } \vee \text{ , } \vee \text{ ident}\end{array}\right\}$

ψ represents any keypunch character other than the quote (").

DEF EXPANSION

defmac expansion	:=	defmac name ∨ (def parlist)
def par list	:=	$\left\{\begin{array}{l}\text{def par} \\ \text{def par list } \vee \text{ , } \vee \text{ def par}\end{array}\right\}$
def par	:=	$\left\{\begin{array}{l}\text{simple def par} \\ \text{special def par}\end{array}\right\}$
simple def par	:=	{any character sequence with balanced bracketing that does not contain delimiting characters ; , or ≡}
special def par		{any character sequence with a single quote represented by two thereof}

EXPRESSIONS

ARITHMETIC EXPRESSIONS

$$\underline{\text{arith exp}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{unary op}} \quad \underline{\text{v}} \\ \phi \end{array} \right\} \quad \underline{\text{infix stuff}}$$

$$\underline{\text{infix stuff}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{arith thing}} \\ \underline{\text{infix stuff}} \quad \underline{\text{v}} \quad \underline{\text{binary op}} \quad \underline{\text{v}} \quad \underline{\text{arith thing}} \end{array} \right\}$$

$$\underline{\text{arith thing}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{item name}} \\ \underline{\text{array reference}} \\ \underline{\text{func call}} \\ \underline{\text{const}} \\ (\quad \underline{\text{v}} \quad \underline{\text{arith exp}} \quad \underline{\text{v}} \quad) \end{array} \right\}$$

$$\underline{\text{unary op}} \quad := \quad \left\{ \begin{array}{l} + \\ - \\ \underline{\text{lno}} \end{array} \right\}$$

$$\underline{\text{binary op}} \quad := \quad \left\{ \begin{array}{l} ** \\ * \\ / \\ + \\ - \\ \underline{\text{lan}} \\ \underline{\text{lor}} \\ \underline{\text{lxr}} \\ \underline{\text{lim}} \\ \underline{\text{lqv}} \end{array} \right\}$$

BOOLEAN EXPRESSIONS

$$\underline{\text{Boolean exp}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{Boolean thing}} \\ \underline{\text{Boolean exp}} \quad \underline{\text{v}} \quad \underline{\text{Boolean op}} \quad \underline{\text{v}} \quad \underline{\text{Boolean thing}} \end{array} \right\}$$

$$\underline{\text{Boolean thing}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{array reference}} \\ \underline{\text{item name}} \\ \underline{\text{relation}} \\ \underline{\text{Boolean const}} \\ \underline{\text{not}} \quad \underline{\text{v}} \quad \underline{\text{Boolean thing}} \\ \underline{\text{func call}} \\ (\quad \underline{\text{v}} \quad \underline{\text{Boolean exp}} \quad \underline{\text{v}} \quad) \end{array} \right\}$$

An item must be declared type B for use as a Boolean operand.

$$\underline{\text{Boolean op}} \quad := \quad \left\{ \begin{matrix} \underline{\text{and}} \\ \underline{\text{or}} \end{matrix} \right\}$$

$$\underline{\text{relation}} \quad := \quad \underline{\text{arith exp}} \; \underline{\vee} \; \underline{\text{relational op}} \; \underline{\vee} \; \underline{\text{arith exp}}$$

$$\underline{\text{relational op}} \quad := \quad \left\{ \begin{matrix} \underline{\text{eq}} \\ \underline{\text{gr}} \\ \underline{\text{ls}} \\ \underline{\text{gq}} \\ \underline{\text{lq}} \\ \underline{\text{nq}} \end{matrix} \right\}$$

CONSTANTS

$$\underline{\text{const}} \quad := \quad \left\{ \begin{matrix} \underline{\text{Boolean const}} \\ \underline{\text{char const}} \\ \underline{\text{integer const}} \\ \underline{\text{real const}} \\ \underline{\text{status const}} \end{matrix} \right\}$$

INTEGER CONSTANTS

$$\underline{\text{integer const}} \quad := \quad \left\{ \begin{matrix} \underline{\text{dec integer}} \\ \underline{\text{octal const}} \\ \underline{\text{hex const}} \\ \underline{\text{status func}} \end{matrix} \right\}$$

The <u>status func</u> is a special form of integer constant defined under status declarations.

$$\underline{\text{dec integer}} \quad := \quad \left\{ \begin{matrix} \underline{\text{dec integer}} \\ \phi \end{matrix} \right\} \underline{\text{digit}}$$

$$\underline{\text{octal const}} \quad := \quad \text{O} \; ' \; \underline{\text{octal stuff}} \; '$$

$$\underline{\text{octal stuff}} \quad := \quad \left\{ \begin{matrix} \underline{\text{octal stuff}} \\ \phi \end{matrix} \right\} \left\{ \begin{matrix} \underline{\text{octal digit}} \\ \underline{\wedge} \end{matrix} \right\}$$

$$\underline{\text{octal digit}} \quad := \quad \left\{ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \right\}$$

HEXADECIMAL CONSTANTS

$$\underline{\text{hex const}} \quad := \quad \text{X} \quad ' \; \underline{\text{hex stuff}} \; '$$

$$\underline{\text{hex stuff}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{hex stuff}} \\ \phi \end{array} \right\} \qquad \left\{ \begin{array}{l} \underline{\text{hex digit}} \\ \underline{\wedge} \end{array} \right\}$$

$$\underline{\text{hex digit}} \quad := \quad \left\{ \begin{array}{c} \underline{\text{digit}} \\ \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \\ \text{E} \\ \text{F} \end{array} \right\}$$

BOOLEAN CONSTANTS

$$\underline{\text{Boolean const}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{true}} \\ \underline{\text{false}} \end{array} \right\}$$

CHARACTER CONSTANTS

$$\underline{\text{char const}} \quad := \quad ' \; \underline{\text{char string}} \; '$$

$$\underline{\text{char string}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{char string}} \; \psi \\ \phi \end{array} \right\}$$

ψ represents any keypunch character.

STATUS CONSTANTS

$$\underline{\text{status const}} \quad := \quad \text{S} \; ' \; \underline{v} \; \underline{\text{status const string}} \; \underline{v} \; '$$

$$\underline{\text{status const string}} \quad := \quad \underline{\text{ident}}$$

REAL CONSTANTS

$$\underline{\text{real const}} \quad := \quad \left\{\begin{array}{l} \underline{\text{integer part}} \\ \phi \end{array}\right\} . \left\{\begin{array}{l} \underline{\text{fraction part}} \\ \phi \end{array}\right\} \left\{\begin{array}{l} \underline{\text{exponent part}} \\ \phi \end{array}\right\}$$

$$\underline{\text{integer part}} \quad := \quad \underline{\text{dec integer}}$$

$$\underline{\text{fraction part}} \quad := \quad \underline{\text{dec integer}}$$

$$\underline{\text{exponent part}} \quad := \quad \left\{\begin{array}{c} \text{D} \\ \text{E} \end{array}\right\} \quad \left\{\begin{array}{cc} + & \underline{\vee} \\ - & \underline{\vee} \\ \phi & \end{array}\right\} \quad \underline{\text{dec integer}}$$

ITEMS

ITEM DECLARATION

$$\underline{\text{item dec}} \quad := \quad \underline{\text{item}} \;\; \underline{\wedge} \;\; \underline{\text{item descr list}} \;\; \underline{\vee} \;\; ;$$

$$\underline{\text{item descr list}} \quad := \quad \left\{\begin{array}{l} \underline{\text{item descr}} \\ \underline{\text{item descr list}} \;\; \underline{\vee} \;\; , \;\; \underline{\vee} \;\; \underline{\text{item descr}} \end{array}\right\}$$

$$\underline{\text{item descr}} \quad := \quad \underline{\text{item name}} \quad \left\{\begin{array}{ll} \underline{\wedge} & \underline{\text{type}} \\ \phi & \end{array}\right\} \quad \underline{\vee} \;\; \underline{\text{item preset}}$$

$$\underline{\text{type}} \quad := \quad \left\{\begin{array}{llllll} \text{U} & & & & & \\ \text{I} & & & & & \\ \text{R} & & & & & \\ \text{B} & & & & & \\ \text{C} & \underline{\vee} & (& \underline{\vee} & \underline{\text{length}} & \underline{\vee} \;\;) \\ \text{S} & \underline{\vee} & : & \underline{\vee} & \underline{\text{status list name}} & \end{array}\right\}$$

U	=	unsigned integer type
I	=	integer type
R	=	real type
B	=	Boolean type
S	=	status type
C	=	character type length is a size subfield in characters or bytes

$$\underline{\text{length}} \quad := \quad \underline{\text{integer const}}$$

ITEM PRESETS

Optionally, the item may be assigned an initial value:

$$\underline{\text{item preset}} \quad := \quad \left\{ \begin{array}{l} = \quad \underline{v} \quad \left\{ \begin{array}{ll} + & \underline{v} \\ - & \underline{v} \\ \phi & \end{array} \right\} \quad \underline{\text{const}} \\ \\ \qquad \phi \end{array} \right\}$$

STATUS DECLARATIONS

SPECIFICATION

$$\underline{\text{status dec}} \quad := \quad \underline{\text{status}} \ \underline{\wedge} \ \underline{\text{status list name}} \ \underline{\wedge} \ \underline{\text{status name list}} \ \underline{v} \ ;$$

$$\underline{\text{status name list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{status value}} \\ \underline{\text{status name list}} \ \underline{v} \ , \ \underline{v} \ \underline{\text{status value}} \end{array} \right\}$$

$$\underline{\text{status value}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{status const string}} \\ \phi \end{array} \right\}$$

STATUS FUNCTION

$$\underline{\text{status func}} \quad := \quad \underline{\text{status list name}} \quad ' \ \underline{v} \ \underline{\text{status const string}} \ \underline{v} \ '$$

ARRAYS

ARRAY DECLARATIONS

$$\underline{\text{array dec}} \quad := \quad \underline{\text{array}} \left\{ \begin{array}{l} \underline{\wedge} \ \underline{\text{array name}} \\ \quad \varnothing \end{array} \right\} \underline{v} \ \underline{\text{array descr}} \ \underline{v} \ ; \ \underline{v} \ \underline{\text{item part}}$$

$$\underline{\text{array descr}} \quad := \quad \left\{ \begin{array}{l} [\ \underline{v} \ \underline{\text{array bounds list}} \ \underline{v} \ ] \\ \quad \varnothing \end{array} \right\} \underline{v} \left\{ \begin{array}{l} \underline{\text{layout}} \\ \varnothing \end{array} \right\} \underline{v} \left\{ \begin{array}{l} \underline{\text{entry size}} \\ \varnothing \end{array} \right\}$$

$$\underline{\text{array bounds list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{bound pair}} \\ \underline{\text{array bounds list}} \ \underline{v} \ , \ \underline{v} \ \underline{\text{bound pair}} \end{array} \right\}$$

$$\underline{\text{bound pair}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{low bound}} \quad \underline{v} \quad : \quad \underline{v} \\ \varnothing \end{array} \right\} \quad \underline{\text{high bound}}$$

$$\underline{\text{low bound}} \quad := \quad \left\{ \begin{array}{ll} + & \underline{} \\ - & \underline{v} \\ \varnothing & \end{array} \right\} \quad \underline{\text{integer const}}$$

$$\underline{\text{high bound}} \quad := \quad \left\{\begin{array}{ll} + & \underline{\vee} \\ - & \underline{\vee} \\ \emptyset & \end{array}\right\} \quad \underline{\text{integer const}}$$

$$\underline{\text{layout}} \quad := \quad \left\{\begin{array}{l} \text{P} \\ \text{S} \end{array}\right\}$$

$$\underline{\text{entry size}} \quad := \quad (\underline{\vee}\ \underline{\text{integer const}}\ \underline{\vee})$$

ARRAY ITEM DECLARATIONS

$$\underline{\text{item part}} \quad := \quad \left\{\begin{array}{l} \underline{\text{begin}} \quad \underline{\wedge} \quad \underline{\text{array item dec list}} \quad \underline{\vee} \quad \underline{\text{end}} \\ \underline{\text{array item dec}} \\ ; \end{array}\right\}$$

$$\underline{\text{array item dec list}} \quad := \quad \left\{\begin{array}{l} \underline{\text{array item dec}} \\ \underline{\text{array item dec list}} \quad \underline{\vee} \quad \underline{\text{array item dec}} \end{array}\right\}$$

$$\underline{\text{array item dec}} \quad := \quad \underline{\text{item}} \quad \underline{\wedge} \quad \underline{\text{array item descr list}} \quad \underline{\vee} \quad ;$$

$$\underline{\text{array item descr list}} \quad := \quad \left\{\begin{array}{l} \underline{\text{array item descr}} \\ \underline{\text{array item descr list}} \quad \vee \quad , \quad \underline{\vee} \quad \underline{\text{array item descr}} \end{array}\right\}$$

$$\underline{\text{array item descr}} \quad := \quad \underline{\text{array item name}} \quad \underline{\vee} \quad \underline{\text{array item specs}} \quad \underline{\vee} \quad \underline{\text{array preset}}$$

$$\begin{array}{l}\underline{\text{array item}} \\ \underline{\text{specs}}\end{array} \quad := \quad \left\{\begin{array}{l} \text{U} \\ \text{I} \\ \text{R} \\ \text{B} \\ \text{C} \\ \text{S}\ \underline{\vee}\ :\ \underline{\vee}\ \underline{\text{status list name}} \\ \emptyset \end{array}\right\} \quad \underline{\vee} \quad \left\{\begin{array}{l} (\underline{\vee}\ \underline{\text{ep}}\ \underline{\vee} \left\{\begin{array}{l} ,\underline{\vee}\ \ \underline{\text{fbit}}\ \underline{\vee} \left\{\begin{array}{l} ,\ \underline{\vee}\ \underline{\text{size}}\ \underline{\vee} \\ \emptyset \end{array}\right\} \\ \emptyset \end{array}\right\}) \\ \emptyset \end{array}\right\}$$

$$\underline{\text{ep}} \quad := \quad \underline{\text{integer const}}$$

$$\underline{\text{fbit}} \quad := \quad \underline{\text{integer const}}$$

$$\underline{\text{size}} \quad := \quad \underline{\text{integer const}}$$

ARRAY PRESETS

$$\underline{\text{array preset}} \quad := \quad \left\{\begin{array}{l} \quad \emptyset \\ = \ \underline{\vee} \ \ \underline{\text{value set}} \end{array}\right\}$$

$$\underline{\text{set sequence}} \quad := \quad \left\{ \begin{array}{l} \phi \\ \underline{\text{set sequence}} \quad \underline{\text{value set}} \\ \underline{\text{set sequence}} \quad \underline{\text{integer const}} \quad \underline{\text{value set}} \end{array} \right\}$$

$$\underline{\text{value set}} \quad := \quad \left\{ \begin{array}{l} [\quad \underline{v} \quad \underline{\text{value list}} \quad \underline{v} \quad] \\ [\quad \underline{v} \quad \underline{\text{set sequence}} \quad \underline{v} \quad] \end{array} \right\}$$

$$\underline{\text{value list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{value}} \\ \underline{\text{integer const}} \quad \underline{v} \quad (\quad \underline{v} \quad \underline{\text{value list}} \quad \underline{v} \quad) \\ \underline{\text{value list}} \quad \underline{v} \quad , \quad \underline{v} \quad \underline{\text{value}} \end{array} \right\}$$

$$\underline{\text{value}} \quad := \quad \left\{ \begin{array}{l} \left\{ \begin{array}{l} + \quad \underline{v} \\ - \quad \underline{v} \\ \phi \end{array} \right\} \underline{\text{const}} \\ \phi \end{array} \right\}$$

ARRAY REFERENCES: SUBSCRIPTS

$$\underline{\text{array reference}} \quad := \quad \underline{\text{array item name}} \quad \underline{v} \quad \underline{\text{subscriptor}}$$

$$\underline{\text{subscriptor}} \quad := \quad [\quad \underline{v} \quad \underline{\text{subscript list}} \quad \underline{v} \quad]$$

$$\underline{\text{subscript list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{subscript}} \\ \underline{\text{subscript list}} \quad \underline{v} \quad , \quad \underline{v} \quad \underline{\text{subscript}} \end{array} \right\}$$

$$\underline{\text{subscript}} \quad := \quad \underline{\text{arith exp}}$$

BASED ARRAYS AND THE P-FUNCTION

$$\underline{\text{based dec}} \quad := \quad \underline{\text{based}} \quad \underline{\wedge} \quad \left\{ \begin{array}{l} \underline{\text{array dec}} \\ \underline{\text{begin}} \quad \underline{\wedge} \quad \underline{\text{array dec list}} \quad \underline{v} \quad \underline{\text{end}} \end{array} \right\}$$

$$\underline{\text{array dec list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{array dec}} \\ \underline{\text{array dec list}} \quad \underline{v} \quad \underline{\text{array dec}} \end{array} \right\}$$

$$\underline{\text{p func}} \quad := \quad P < \quad \underline{v} \quad \underline{\text{based array name}} \quad \underline{v} \quad >$$

FUNCTIONS

FUNCTION CALLS

$$\underline{\text{func call}} \quad := \quad \left\{ \begin{array}{ll} \underline{\text{func name}} & \left\{ \begin{array}{l} \underline{\text{arguments}} \\ \phi \end{array} \right\} \\ \underline{\text{bead func}} & \\ \underline{\text{loc func}} & \\ \underline{\text{p func}} & \\ \underline{\text{abs func}} & \end{array} \right\}$$

$$\underline{\text{arguments}} \quad := \quad (\quad \underline{\text{v}} \quad \underline{\text{actual par list}} \quad \underline{\text{v}} \quad)$$

$$\underline{\text{actual par list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{actual par}} \\ \underline{\text{actual par list}} \quad \underline{\text{v}} \quad , \quad \underline{\text{v}} \quad \underline{\text{actual par}} \end{array} \right\}$$

BEAD FUNCTION

$$\underline{\text{bead func}} \quad := \quad \left\{ \begin{array}{c} \text{B} \\ \text{C} \end{array} \right\} < \quad \underline{\text{v}} \quad \underline{\text{arith exp}} \quad \underline{\text{v}} \left\{ \begin{array}{l} , \quad \underline{\text{v}} \quad \underline{\text{arith exp}} \quad \underline{\text{v}} \\ \quad \phi \end{array} \right\} > \underline{\text{v}} \quad \underline{\text{data}}$$

$$\underline{\text{data}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{item name}} \\ \underline{\text{array reference}} \end{array} \right\}$$

INTRINSIC LOC FUNCTION

$$\underline{\text{loc func}} \quad := \quad \underline{\text{loc}} \quad \underline{\text{v}} \quad (\quad \underline{\text{v}} \left\{ \begin{array}{ll} \underline{\text{item name}} & \\ \underline{\text{array reference}} & \\ \underline{\text{proc name}} & \\ \underline{\text{func name}} & \\ \underline{\text{switch name}} & \\ \underline{\text{label name}} & \\ \underline{\text{array name}} & \left\{ \begin{array}{l} \underline{\text{v}} \quad \underline{\text{subscriptor}} \\ \phi \end{array} \right\} \end{array} \right\} \underline{\text{v}} \quad)$$

INTRINSIC ABS FUNCTION

$$\underline{\text{abs func}} \quad := \quad \underline{\text{abs}} \quad \underline{\text{v}} \quad (\quad \underline{\text{v}} \quad \underline{\text{arith exp}} \quad \underline{\text{v}} \quad)$$

VALUE ASSIGNMENT

$$\underline{\text{replacement statement}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{sink}} \\ \underline{\text{func name}} \end{array} \right\} \; \underline{\text{v}} \; = \; \underline{\text{v}} \; \underline{\text{source}} \; \underline{\text{v}} \; ;$$

$$\underline{\text{exchange statement}} \quad := \quad \underline{\text{sink}} \; \underline{\text{v}} \; = \; = \; \underline{\text{v}} \; \underline{\text{sink}} \; \underline{\text{v}} \; ;$$

$$\underline{\text{sink}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{item name}} \\ \underline{\text{array reference}} \\ \underline{\text{p func}} \\ \underline{\text{bead func}} \end{array} \right\}$$

$$\underline{\text{source}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{arith exp}} \\ \underline{\text{Boolean exp}} \end{array} \right\}$$

FLOW OF CONTROL

LABEL DECLARATION

$$\underline{\text{label dec}} \quad := \quad \underline{\text{label name}}:$$

$$\underline{\text{labeled statement}} \quad := \quad \underline{\text{label dec}} \; \left\{ \begin{array}{l} \underline{\text{v}} \; \underline{\text{statement}} \\ \phi \end{array} \right\}$$

SWITCH DECLARATION

$$\underline{\text{switch dec}} \quad := \quad \underline{\text{switch}} \; \underline{\wedge} \; \underline{\text{switch name}} \; \underline{\text{v}} \; \underline{\text{switch specs}} \; \underline{\text{v}} \; ;$$

$$\underline{\text{switch specs}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{switch list}} \\ \\ : \; \underline{\text{v}} \; \underline{\text{status list name}} \; \underline{\text{v}} \; \underline{\text{switch order}} \end{array} \right\}$$

$$\underline{\text{switch list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{switch point}} \\ \\ \underline{\text{switch list}} \; \underline{\text{v}} \; , \; \underline{\text{v}} \; \underline{\text{switch point}} \end{array} \right\}$$

$$\underline{\text{switch point}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{label name}} \\ \phi \end{array} \right\}$$

$$\underline{\text{switch order}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{order pair}} \\ \\ \underline{\text{switch order}} \; \underline{\text{v}} \; , \; \underline{\text{v}} \; \underline{\text{order pair}} \end{array} \right\}$$

$$\underline{\text{order pair}} \quad := \quad \underline{\text{label name}}\ \ \underline{\vee}\ \ :\ \ \underline{\vee}\ \ \underline{\text{status const string}}$$

GOTO STATEMENT

$$\underline{\text{goto statement}} \quad := \quad \underline{\text{goto}}\ \ \underline{\wedge}\ \ \left\{\begin{array}{l}\underline{\text{label name}}\\ \underline{\text{switch name}}\ \ \underline{\vee}\ \ [\ \ \underline{\vee}\ \ \underline{\text{arith exp}}\ \ \underline{\vee}\ \ ]\end{array}\right\}\ \underline{\vee}\ \ ;$$

IF STATEMENT

$$\underline{\text{if statement}} \quad := \quad \underline{\text{if clause}}\ \ \underline{\vee}\ \ \underline{\text{statement}}\ \ \left\{\begin{array}{l}\underline{\vee}\ \ \underline{\text{else part}}\\ \phi\end{array}\right\}$$

$$\underline{\text{if clause}} \quad := \quad \underline{\text{if}}\ \ \underline{\vee}\ \ \underline{\text{Boolean exp}}\ \ \underline{\vee}\ \ \underline{\text{then}}$$

$$\underline{\text{else part}} \quad := \quad \underline{\text{else}}\ \ \underline{\vee}\ \ \underline{\text{statement}}$$

FOR STATEMENT

$$\underline{\text{for statement}} \quad := \quad \underline{\text{for clause}}\ \ \underline{\vee}\ \ \underline{\text{statement}}$$

$$\underline{\text{for clause}} \quad := \quad \underline{\text{for}}\ \ \underline{\wedge}\ \ \underline{\text{item name}}\ \ \underline{\vee}\ \ =\ \ \underline{\vee}\ \ \underline{\text{loop control}}\ \ \underline{\vee}\ \ \underline{\text{do}}$$

$$\underline{\text{loop control}} \quad := \quad \underline{\text{initial value}}\ \ \left\{\begin{array}{l}\underline{\vee}\ \ \underline{\text{step part}}\ \ \left\{\begin{array}{l}\underline{\vee}\ \ \underline{\text{while part}}\\ \underline{\vee}\ \ \underline{\text{until part}}\\ \phi\end{array}\right\}\\ \underline{\vee}\ \ \underline{\text{while part}}\\ \phi\end{array}\right\}$$

$$\underline{\text{initial value}} \quad := \quad \underline{\text{arith exp}}$$

$$\underline{\text{step part}} \quad := \quad \underline{\text{step}}\ \ \underline{\vee}\ \ \underline{\text{arith exp}}$$

$$\underline{\text{until part}} \quad := \quad \underline{\text{until}}\ \ \underline{\vee}\ \ \underline{\text{arith exp}}$$

$$\underline{\text{while part}} \quad := \quad \underline{\text{while}}\ \ \underline{\vee}\ \ \underline{\text{Boolean exp}}$$

TEST STATEMENT

$$\underline{\text{test statement}} \quad := \quad \underline{\text{test}}\ \ \left\{\begin{array}{l}\underline{\wedge}\ \ \underline{\text{item name}}\\ \phi\end{array}\right\}\ \underline{\vee}\ \ ;$$

PROCEDURES
PROCEDURE CALL STATEMENT

$$\underline{\text{proc call statement}} \quad := \quad \underline{\text{proc name}}\ \ \left\{\begin{array}{l}\underline{\vee}\ \ \underline{\text{arguments}}\\ \phi\end{array}\right\}\ \ \underline{\vee}\ \ ;$$

RETURN STATEMENT

$\underline{\text{return statement}} \quad := \quad \underline{\text{return}}\ \ \underline{\text{v}}\ \ ;$

STOP STATEMENT

$\underline{\text{stop statement}} \quad := \quad \underline{\text{stop}}\ \ \underline{\text{v}}\ \ ;$

SUBPROGRAM DECLARATIONS

$$\underline{\text{subprogram dec}} \quad := \quad \left\{\begin{matrix} \underline{\text{proc dec}} \\ \underline{\text{func dec}} \end{matrix}\right\}$$

$$\underline{\text{proc dec}} \quad := \quad \underline{\text{proc dec clause}}\ \ \underline{\text{v}}\ \ \underline{\text{dec list}}\ \ \underline{\text{v}}\ \ \underline{\text{statement}}$$

$$\underline{\text{func dec}} \quad := \quad \underline{\text{func dec clause}}\ \ \underline{\text{v}}\ \ \underline{\text{dec list}}\ \ \underline{\text{v}}\ \ \underline{\text{statement}}$$

$$\underline{\text{proc dec clause}} \quad := \quad \underline{\text{proc}}\ \ \underline{\wedge}\ \ \underline{\text{proc name}} \left\{\begin{matrix} \underline{\text{v}}\ \ (\ \ \underline{\text{v}}\ \ \underline{\text{formal par list}}\ \ \underline{\text{v}}\ \ ) \\ \phi \end{matrix}\right\} \underline{\text{v}}\,;$$

$$\underline{\text{formal par list}} \quad := \quad \left\{\begin{matrix} \underline{\text{formal par}} \\ \underline{\text{formal par list}}\ \ \underline{\text{v}}\ \ ,\ \ \underline{\text{v}}\ \ \underline{\text{formal par}} \end{matrix}\right\}$$

$$\underline{\text{func dec clause}} \quad := \quad \underline{\text{func}}\ \ \underline{\wedge}\ \ \underline{\text{func name}} \left\{\begin{matrix} \left\{\begin{matrix} \underline{\text{v}} \\ \underline{\text{v}}\ \ (\ \ \underline{\text{v}}\ \ \underline{\text{formal par list}}\ \ \underline{\text{v}}\ \ ) \end{matrix}\right\} \\ \left\{\begin{matrix} \underline{\wedge} \\ \phi \end{matrix}\right\} \underline{\text{type}} \end{matrix}\right\} \left\{\begin{matrix} \underline{\text{v}}\ \underline{\text{type}} \\ \phi \end{matrix}\right\} \underline{\text{v}}\,;$$

$$\underline{\text{dec list}} \quad := \quad \left\{\begin{matrix} \underline{\text{declaration}} \\ \underline{\text{dec list}}\ \ \underline{\text{v}}\ \ \underline{\text{declaration}} \\ \phi \end{matrix}\right\}$$

LABELS AND PARAMETERS

FORMAL LABEL DECLARATIONS

$$\underline{\text{formal label dec}} \quad := \quad \underline{\text{label}}\ \ \underline{\wedge}\ \ \underline{\text{label name list}}\ \ \underline{\text{v}}\ \ ;$$

$$\underline{\text{label name list}} \quad := \quad \left\{\begin{matrix} \underline{\text{label name}} \\ \underline{\text{label name list}}\ \ \underline{\text{v}}\ \ ,\ \ \underline{\text{v}}\ \ \underline{\text{label name}} \end{matrix}\right\}$$

FORMAL PARAMETERS

$$\underline{\text{formal based dec}} \quad := \quad \underline{\text{based dec}}$$

$$\underline{\text{formal item dec}} \quad := \quad \underline{\text{item dec}}$$

$$\underline{\text{formal array dec}} \quad := \quad \underline{\text{array dec}}$$

$$\underline{\text{formal proc dec}} \quad := \quad \underline{\text{fprc}} \;\; \underline{\wedge} \;\; \underline{\text{formal proc name}} \;\; \underline{\vee} \;\; ;$$

$$\underline{\text{formal func dec}} \quad := \quad \underline{\text{func}} \;\; \underline{\wedge} \;\; \underline{\text{formal func name}} \left\{ \begin{matrix} \underline{\wedge} \;\; \underline{\text{type}} \\ \phi \end{matrix} \right\} \;\; \underline{\vee} \;\; ;$$

$$\underline{\text{value par}} \quad := \quad (\underline{\vee} \; \underline{\text{formal item name}} \; \underline{\vee})$$

$$\underline{\text{formal par}} \quad := \quad \left(\begin{matrix} \underline{\text{formal based name}} \\ \underline{\text{formal item name}} \\ \underline{\text{formal array name}} \\ \underline{\text{formal proc name}} \\ \underline{\text{formal func name}} \\ \underline{\text{label name}} \\ \underline{\text{value par}} \end{matrix} \right)$$

ACTUAL PARAMETERS

$$\underline{\text{actual par}} \quad := \quad \left(\begin{matrix} \underline{\text{item name}} \\ \underline{\text{array name}} \\ \underline{\text{proc name}} \\ \underline{\text{func name}} \\ \underline{\text{label name}} \\ \underline{\text{arith exp}} \\ \underline{\text{Boolean exp}} \\ \underline{\text{p func}} \end{matrix} \right)$$

ENTRIES

$$\underline{\text{entry dec}} \quad := \quad \underline{\text{entry}} \;\; \underline{\wedge} \left\{ \begin{matrix} \underline{\text{proc dec clause}} \\ \underline{\text{func dec clause}} \end{matrix} \right\} \quad ;$$

COMMON

$$\underline{\text{common dec}} \quad := \quad \underline{\text{common}} \left\{\begin{matrix} \underline{\wedge} & \underline{\text{common name}} \\ \phi & \end{matrix}\right\} \underline{\vee} \; ; \; \underline{\vee} \left\{\begin{matrix} \underline{\text{data dec}} \\ \underline{\text{begin}}\ \underline{\vee}\ \underline{\text{data dec list}}\ \underline{\vee}\ \underline{\text{end}} \end{matrix}\right\}$$

$$\underline{\text{data dec list}} \quad := \left\{\begin{matrix} \underline{\text{data dec}} \\ \underline{\text{data dec list}}\ \ \underline{\vee}\ \ \underline{\text{data dec}} \end{matrix}\right\}$$

$$\underline{\text{data dec}} \quad := \left\{\begin{matrix} \underline{\text{item dec}} \\ \underline{\text{array dec}} \end{matrix}\right\}$$

EXTERNALS

XREF (EXTERNAL REFERENCE) DECLARATIONS

$$\underline{\text{xref dec}} \quad := \quad \underline{\text{xref}}\ \ \underline{\wedge}\ \ \underline{\text{xdec part}}$$

$$\underline{\text{xdec part}} \quad := \left\{\begin{matrix} \underline{\text{begin}}\ \ \underline{\wedge}\ \ \underline{\text{xdec list}}\ \ \underline{\vee}\ \ \underline{\text{end}} \\ \underline{\text{xdec}} \end{matrix}\right\}$$

$$\underline{\text{xdec list}} \quad := \left\{\begin{matrix} \underline{\text{xdec}} \\ \underline{\text{xdec list}}\ \ \underline{\vee}\ \ \underline{\text{xdec}} \end{matrix}\right\}$$

$$\underline{\text{xdec}} \quad := \left\{\begin{matrix} \underline{\text{item dec}} \\ \underline{\text{array dec}} \\ \underline{\text{proc heading}} \\ \underline{\text{func heading}} \\ \underline{\text{formal label dec}} \\ \underline{\text{switch dec}} \\ \underline{\text{formal switch dec}} \\ \underline{\text{based dec}} \end{matrix}\right\}$$

$$\underline{\text{formal label dec}} \quad := \quad \underline{\text{label}}\ \underline{\wedge}\ \underline{\text{label name list}}\ \underline{\vee}\ ;$$

$$\underline{\text{label name list}} \quad := \left\{\begin{matrix} \underline{\text{label name}} \\ \underline{\text{label name list}}\ \underline{\vee}\ ,\ \underline{\vee}\ \underline{\text{label name}} \end{matrix}\right\}$$

$$\underline{\text{formal switch dec}} \quad := \quad \underline{\text{switch}}\ \underline{\wedge}\ \underline{\text{switch name list}}\ \underline{\vee}\ ;$$

$$\underline{\text{switch name list}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{switch name}} \\ \underline{\text{switch name list}} \; \underline{\vee} \; , \; \underline{\vee} \; \underline{\text{switch name}} \end{array} \right\}$$

$$\underline{\text{proc heading}} \quad := \quad \underline{\text{proc}} \; \underline{\wedge} \; \underline{\text{proc name}} \; \underline{\vee} \; ;$$

$$\underline{\text{func heading}} \quad := \quad \underline{\text{func}} \; \underline{\wedge} \; \underline{\text{func name}} \left\{ \begin{array}{l} \underline{\wedge} \\ \phi \end{array} \; \underline{\text{type}} \right\} \quad \underline{\vee} \; ;$$

XDEF (EXTERNAL DEFINITION) DECLARATIONS

$$\underline{\text{xdef dec}} \quad := \quad \underline{\text{xdef}} \; \underline{\wedge} \; \underline{\text{xdec part}}$$

PROGRAMS

PROGRAM STRUCTURE

$$\underline{\text{program}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{program head}} \\ \underline{\text{subprogram dec}} \end{array} \right\} \quad \underline{\vee} \; \underline{\text{term}}$$

$$\underline{\text{program head}} \quad := \quad \left\{ \begin{array}{lll} \underline{\text{prgm dec}} & & \\ \underline{\text{program head}} & \underline{\vee} & \underline{\text{declaration}} \\ \underline{\text{program head}} & \underline{\vee} & \underline{\text{statement}} \end{array} \right\}$$

$$\underline{\text{prgm dec}} \quad := \quad \underline{\text{prgm}} \; \underline{\wedge} \; \underline{\text{program name}} \; ;$$

COMPOUND STATEMENTS

$$\underline{\text{compound statement}} \quad := \quad \left\{ \begin{array}{lll} \underline{\text{compound head}} & \underline{\vee} & \underline{\text{end}} \\ \underline{\text{compound head}} & \underline{\vee} & \underline{\text{spend}} \end{array} \right\}$$

$$\underline{\text{compound head}} \quad := \quad \left\{ \begin{array}{lll} \underline{\text{begin}} & & \\ \underline{\text{spbegin}} & & \\ \underline{\text{compound head}} & \underline{\vee} & \underline{\text{statement}} \\ \underline{\text{compound head}} & \underline{\vee} & \underline{\text{declaration}} \end{array} \right\}$$

CONTROL STATEMENT

$$\underline{\text{control statement}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{control}} \; \underline{\wedge} \; \underline{\text{control word}} \; \underline{\vee} \; ; \\ \underline{\text{control}} \; \underline{\wedge} \; \underline{\text{conditional phrase}} \; \underline{\vee} \; ; \end{array} \right\}$$

$$\underline{\text{conditional phrase}} \quad := \quad \underline{\text{condition word}} \; \underline{\wedge} \; \underline{\text{condition params}}$$

$$\underline{\text{condition params}} \quad := \quad \left\{ \begin{array}{l} \underline{\text{constant}} \\ \underline{\text{constant}} \; \underline{\vee} \; , \; \underline{\vee} \; \underline{\text{constant}} \end{array} \right\}$$

$$\underline{\text{condition word}} \quad := \quad \left\{\begin{array}{l}\underline{\text{ifeq}}\\ \underline{\text{ifne}}\\ \underline{\text{ifls}}\\ \underline{\text{iflq}}\\ \underline{\text{ifgq}}\\ \underline{\text{ifgr}}\end{array}\right\}$$

$$\underline{\text{control word}} \quad := \quad \left\{\begin{array}{l}\underline{\text{eject}}\\ \underline{\text{list}}\\ \underline{\text{nolist}}\\ \underline{\text{objlst}}\\ \underline{\text{pack}}\\ \underline{\text{preset}}\\ \underline{\text{fi}}\end{array}\right\}$$

$$\begin{array}{llllll}
\underline{\text{ifeq}} & := & \underline{\text{mark}} & \rfloor\ \text{IFEQ} & \lfloor & \underline{\text{mark}}\\
\underline{\text{ifne}} & := & \underline{\text{mark}} & \rfloor\ \text{IFNE} & \lfloor & \underline{\text{mark}}\\
\underline{\text{ifls}} & := & \underline{\text{mark}} & \rfloor\ \text{IFLS} & \lfloor & \underline{\text{mark}}\\
\underline{\text{iflq}} & := & \underline{\text{mark}} & \rfloor\ \text{IFLQ} & \lfloor & \underline{\text{mark}}\\
\underline{\text{ifgq}} & := & \underline{\text{mark}} & \rfloor\ \text{IFGQ} & \lfloor & \underline{\text{mark}}\\
\underline{\text{ifgr}} & := & \underline{\text{mark}} & \rfloor\ \text{IFGR} & \lfloor & \underline{\text{mark}}\\
\underline{\text{eject}} & := & \underline{\text{mark}} & \rfloor\ \text{EJECT} & \lfloor & \underline{\text{mark}}\\
\underline{\text{list}} & := & \underline{\text{mark}} & \rfloor\ \text{LIST} & \lfloor & \underline{\text{mark}}\\
\underline{\text{objlst}} & := & \underline{\text{mark}} & \rfloor\ \text{OBJLST} & \lfloor & \underline{\text{mark}}\\
\underline{\text{pack}} & := & \underline{\text{mark}} & \rfloor\ \text{PACK} & \lfloor & \underline{\text{mark}}\\
\underline{\text{preset}} & := & \underline{\text{mark}} & \rfloor\ \text{PRESET} & \lfloor & \underline{\text{mark}}
\end{array}$$

$$\underline{\text{fi}} \quad := \quad \left\{\begin{array}{llll}\underline{\text{mark}} & \rfloor\ \text{FI} & \lfloor & \underline{\text{mark}}\\ \underline{\text{mark}} & \rfloor\ \text{ENDIF} & \lfloor & \underline{\text{mark}}\end{array}\right\}$$

The above are not reserved words.

$$\underline{\text{declaration}} \quad := \left\{ \begin{array}{l} \underline{\text{array dec}} \\ \underline{\text{based dec}} \\ \underline{\text{common dec}} \\ \underline{\text{def dec}} \\ \underline{\text{entry dec}} \\ \underline{\text{func dec}} \\ \underline{\text{item dec}} \\ \underline{\text{label dec}} \\ \underline{\text{proc dec}} \\ \underline{\text{status dec}} \\ \underline{\text{switch dec}} \\ \underline{\text{xdef dec}} \\ \underline{\text{xref dec}} \\ \underline{\text{formal array dec}} \\ \underline{\text{formal based dec}} \\ \underline{\text{formal func dec}} \\ \underline{\text{formal item dec}} \\ \underline{\text{formal label dec}} \\ \underline{\text{formal proc dec}} \end{array} \right\}$$

$$\underline{\text{statement}} \quad := \left\{ \begin{array}{l} \underline{\text{compound statement}} \\ \underline{\text{exchange statement}} \\ \underline{\text{for statement}} \\ \underline{\text{goto statement}} \\ \underline{\text{if statement}} \\ \underline{\text{labeled statement}} \\ \underline{\text{proc call statement}} \\ \underline{\text{replacement statement}} \\ \underline{\text{return statement}} \\ \underline{\text{stop statement}} \\ \underline{\text{test statement}} \end{array} \right\}$$

OUTPUT FROM COMPILATION E

When the optional list parameters L, R, and X are selected on the SYMPL compiler call statement, the compiler outputs a normal source program listing with diagnostic messages following the listing, a cross-reference table, and a storage map which are useful debugging aids. With the X or R option, a common block list is output also.

Below is a source listing of a SYMPL program (intentionally coded with errors) along with its storage map and cross-reference table.

SOURCE LISTING

The user can request a printed listing of any source program or source procedure compiled by specifying the optional list parameter L on the SYMPL compiler call statement. Each line in the listing corresponds to one line in the source deck. The compiler assigns a line number to each source line in a deck beginning at 0001 which appears on the left-hand side of the source listing (column 1).

Column 2 defines the BEGIN ... END nesting levels, a minus sign in this column indicates the line contains code suppressed during conditional compilation.

The diagnostic number is displayed in column 3. When a diagnostic number appears in this column, the numbering sequence in column 1 is interrupted by a sequence of ****, a diagnostic flag.

Column 4 displays the ID or declaration causing the message.

Column 5 displays the source program listing.

After the last source line, the compiler displays a summation of all compiler infringements and displays this number; in addition the compiler lists each infringement along with its message number (in ascending order) and appropriate definition.

```
         SORT100    PROCEDURE            * SOURCE LISTING *            SYMPL 1.0 (072771)   10/18/71

                             ⑤
   0001.                     PRGM SORT100  ;   ≡  ONE-HUNDRED WORD SYMPL SORT POUTINE   ≡
   0002.                     BASED ARRAY AA[99]  ;
   0003.                          ITEM X  ;
   0004.                     XDEF PROC SCRTER  ;
   0005.                     ARRAY TOBESORTED [99]  ;
   0006.     ③       ④            ITEM T  ;
   0007.                     P<AA> = LOC(TOBESORTED)  ;
*******      77     SORTER
*******      16
*******      28
   0008. ②                        SOPTER  P<AA>   ;
   0009.                          PROC SCRTER(SORT)  ;
   0010. B 1                      BEGIN
   0011.                               BASED ARRAY SORT[99]  ;
   0012.                               ITEM VALUE  ;
   0013.                               TTEM FLAG I =0 ;
*******       3     I
*******     105
*******      90
*******      16
*******      88
   0014.                            L1:  FOR I=0 STEP 1 UNTIL 98 DO
*******       3     I
*******       3     I
*******     106
*******      91
*******      16
*******      89
   0015.                               IF VALUE[I+1] GR VALUE[I] THEN
   0016. B 2                           BEGIN
*******       3     I
*******       3     I
*******      97
*******      16
*******      28
   0017.                               VALUE[I+1] == VALUE[I]
   0018.                               FLAG = 1  ;
   0019. E 2                           END
   0020.                          IF FLAG EQ 0 THEN
   0021.                     RETURN  ;
   0022.                               FLAG = 0  ;
   0023.                               GCTO L1  ;
   0024. E 1                      END
   0025.                     TERM
** 19 DIAGNOSTIC MESSAGE(S)
*******       3               UNDECL ID DELETED
*******      16               CRUD AT START OF STMT DELETED
*******      28               SEMI ACCEPTED AS NULL STMT
*******      77               ILL LABEL/PROC ID USE DELETED
*******      88               YOUR -DO- HAS BEEN FOUND
*******      89               THE -THEN- HAS BEEN FOUND
*******      90               MISSING -DC-
*******      91               MISSING -THEN-
*******      97               BAD REPL STMT DELETED
*******     105               FOR STMT: INDUCTION ID ERR
*******     106               -IF- EXPR ERR
PROGRAM LENGTH      000162B WORDS
```

Figure E-1. SYMPL Program Source Listing

STORAGE MAP AND CROSS-REFERENCE TABLE

The storage map and cross-reference table is a dictionary of all programmer created declarations appearing in the source program, with the properties of each declaration and references to them listed by source line number (cross-reference table only). The storage map and cross-reference table begin on a separate page following the source listing of the program and error message dictionary.

STORAGE MAP

1	NAME	First ten characters only of declarations are printed.
2	TYPE	Defines the name as one of the following types:

ARYITM	Array item
COMMON	Common block
ITEM	Item
FUNC	Function
PROC	Procedure
LABEL	Label
B.ARRY	Based array
ARRAY	Array
PROGRAM	Program

3	M	Mode of data representation

B	Boolean
C	Character
I	Integer
P	Parallel (arrays only)
S	Status (serial if type - array)
U	Unsigned integer
X	External

4	LOC	Octal address relative to start of routine; if followed by C, LOC is relative to start of common block. If type = ARYITM, LOC refers to first occurrence of item.
5	FBIT	First bit, numbered from 0 to 59, left to right.
6	NUM	Number of bits; if MODE = C, number of bytes.

```
       SORT100    PROCEDURE              * STORAGE MAP *                   SYMPL 1.0 (072771)   10/18/71

  ①            ②    ③   ④         ⑤    ⑥
NAME:C(10) TYPE    M LOC      FBIT NUM    NAME:C(10) TYPE     M LOC     FBIT NUM    NAME:C(10) TYPE    M LOC     FBIT NUM

AA         B.ARRY P 00000C                FLAG       ITEM     I 000146     0  60    L1         LABEL     000152
SORT       B.ARRY P 000145                SORTER     PROC       00(150              SORT100    PROGRM    000157
SYS=       PROC   X 000000                T          ARYITM   I 000001     0  60    TOBESORTED ARRAY   P 000001
VALUE      ARYITM I 000000      0   60    X          ARYITM   I 000000     0  60
```

Figure E-2. Storage Map Listing

CROSS-REFERENCE TABLE

1 NAME — First ten characters only of declarations are printed.

2 TYPE — Defines the name as one of the following types:

ARYITM	Array item
COMMON	Common block
ITEM	Item
FUNC	Function
PROC	Procedure
LABEL	Label
B.ARRY	Based array
STSCON	Status constant
DEFINE	DEF
STSLST	Status list
PROGRAM	Program
ARRAY	Array

3 M — Mode of data representation

B	Boolean
C	Character
I	Integer
P	Parallel (arrays only)
S	Status (serial if type = array)
U	Unsigned integer
X	External

4 DEF — Line number in source listing where declaration is defined; if followed by C, declaration is in common block.

5 SCOPE — Name of outermost procedure within which declaration occurs; if type = STSCON, SCOPE is the name of the status list of which the item is a member.

6 SET/USED — Source listing line numbers of references to NAME, * indicates use as other than left-hand side of the replacement statement.

```
    SORT100   PROCEDURE            * CROSS REFERENCE *          SYMPL 1.0 (072771)  10/18/71

       ①          ②      ③    ④       ⑤          ⑥
    NAME:C(10) TYPE      M   DEF     SCOPE      SET/USED ( USED INDICATED BY * )

    AA         B.ARRY    P     2     SORT100          7       8*
    FLAG       ITEM      I    13     SORTER          18      22       20*
    L1         LABEL          14     SORTER          23*
    TOBESORTED ARRAY     P     5     SORT100          7*
    VALUE      ARYITM    I    12     SORTER          17      15*

*****      47500 WORDS WERE USED     *****
```

Figure E-3. Cross-Reference Table

OBJECT TIME OUTPUT F

Output for debugging purposes, both initial testing and maintenance, may be performed through the FTN library routines. Linkage is the SYMPL library routine SYMIO. The FTN library routines must be initialized by a FTN main program.

The following declarations are required:

```
XREF BEGIN PROC PRINT;
           PROC LIST;
           PROC ENDL; END
```

Should a conflict in nomenclature arise, these routines can be called PRINT$, LIST$, ENDL$.

PRINT,PRINT$	PRINT (character string); character string must be a FORTRAN format string; it is used to format arguments of LIST.
LIST,LIST$	LIST (argument); argument may be an item, expression, subscripted array item, etc. Its format on file OUTPUT is determined by the next format item in the PRINT string.
END,ENDL$	ENDL; This call must be made to process right alignments and to ensure transmission of the last LIST argument.
PRINTFL	PRINTFL (character string, lfn) lfn must represent an existing FET, probably an XREF ARRAY. This file is used instead of OUTPUT when LIST arguments are transmitted.

Example 1:

```
XREF BEGIN PROC LIST; PROC PRINT; PROC ENDL; END
      .
      .
      .
PRINT ('(1X,*VALUE OF I = *,I3,/)');
LIST (I);
ENDL;
```

This example is equivalent in FORTRAN to:

```
      PRINT 99,I
   99 FORMAT (1X,*VALUE OF I = *,I3,/)
```

Example 2:

```
   XREF BEGIN PROC PRINT; PROC LIST; PROC ENDL; END
        .
        .
        .
   CNTR = LOC(ADDR);
   PRINT('(1X,06/4030)');
   ITEM I; FOR I=0 STEP 4 UNTIL N DO
        BEGIN LIST(CNTR);
               ITEM K; FOR K=0 STEP 1 UNTIL 3 DO LIST (DITM K+1);
               CNTR=CNTR + 4;
        END ≡I≡
   ENDL;
        .
        .
        .
```

This example is equivalent in FORTRAN to:

```
        CNTR = LOC(ADDR)
        DO 1 I = 1,N-1,4
        PRINT 100,CNTR, (DITM(K+I-2),K=1,3)
      1 CNTR = CNTR+4

   100  FORMAT (1X,06/4030)
```

PROGRAMMING SUGGESTIONS G

COMPILER

Space required for compilation is proportional to the number of symbols in the source program. Five words of core are dedicated to each symbol in the program, in the form of a symbol table entry.

Time required for compilation is proportional to the size of the object program, in terms of the amounts of syntax to be scanned. Although data declarations do not generate code, they use significant amounts of compiler time, especially data presets.

Compilation time may be further reduced by judicious use of the compiler options such as object code and cross reference listings.

DEF declarations can increase readability of SYMPL source programs and facilitate changes to them. However, DEF declarations and expansions increase compilation time accordingly.

OBJECT CODE

SUBSCRIPTS

Code produced by referencing subscripted variables can be affected by the means of expressing the subscript. For example, an integer constant can be partially evaluated at compile time so that one instruction is required to access an array item (given the item is a full word); but a scalar integer variable requires four instructions to access the item. Thus, a reference to A [3] requires one instruction for a serial array; but A [I] where I=3, requires four instructions to retrieve the same item.

ARRAYS

Parallel arrays (default case) are accessed more efficiently than serial arrays, when an array entry exceeds one word. For arrays with one-word entries, no difference in object code speed or space is apparent. Parallel arrays, rather than serial, should be used when possible. Fixed arrays are accessed more efficiently than based arrays, which require a level of indirectness to access an entry. Whenever possible, fixed arrays should be used.

DATA TYPES

If an array item is a full 60-bit word, access does not depend upon its type. For items which are not 60-bit words, however, type and bit position assignment affect the code required to access them, as follows:

Signed integers are accessed more efficiently than unsigned integers if the item is not exactly 18 bits long. If the item is 18 bits long, the SXi instruction is used to access both signed and unsigned integers, and the time required is the same. Signed integer items are accessed more efficiently if they are the leftmost bits of a word. Unsigned integer items are accessed more efficiently if they are the rightmost bits of a word. Boolean items are most efficiently accessed by allocating the whole word or the minimum required bits starting with the leftmost bit.

FOR LOOPS

The break-even point in code generation between hand-coded and FOR loop code is 3-4 iterations. Of the following sequences, the second generates fewer instructions and runs faster.

```
FOR I=0 STEP UNTIL 2 DO
PWORD [I] = 0;                      ≡ CODE SEQUENCE 1 ≡

PWORD [0] = 0;                      ≡ CODE SEQUENCE 2 ≡
PWORD [1] = 0;
PWORD [2] = 0;                      ≡ END SEQUENCE 2 ≡
```

If four or more items were being set by the above sequence, the loop would have required less code and would execute in less time.

In general, the less source code in the FOR statement, the faster it will run. Of the following code sequences, the second is faster; since the loop limit is computed and the value stored only once.

```
FOR I = 0 STEP 1 UNTIL B/C DO
    PWORD [I] = K**J;               ≡ CODE SEQUENCE 1 ≡

A = B/C;                            ≡ CODE SEQUENCE 2 ≡
D = K**J;
FOR I = 0 STEP 1 UNTIL A DO
    PWORD [I] = D;                  ≡ END SEQUENCE 2 ≡
```

One exception is that FOR loop execution time can be reduced with more source code as in the following example where the second sequence would be faster even though more code would be generated.

```
FOR I=0 STEP 1 UNTIL 89 DO
    PWORD [I] = 0;                  ≡ CODE SEQUENCE 1 ≡

FOR I = 0 STEP 3 UNTIL 89 DO        ≡ CODE SEQUENCE 2 ≡
    BEGIN
      PWORD [I] = 0;
      PWORD [I+1] = 0;
      PWORD [I+2] = 0;
    END                             ≡ END SEQUENCE 2 ≡
```

DATA CONVERSION

Integer-to-character conversion is byte-oriented while the character-to-integer conversion is word-oriented. When an integer item is converted to character mode, the least significant 6-bit byte is left justified and blank filled in the character field; yet, character-to-integer conversion is performed by right justifying the right end of the last word of the character item and zero filling it on the left. Character field definitions may cross word boundaries but character operations may not.

The conversions may be circumvented by the use of bit bead functions. For example, B <0,60> FLTINGPT = INTEGER; would cause the integer to be stored in the floating point item without conversion. "B <0,60> CHARACTER = INTEGER;" also would cause the full word to be stored in CHARACTER, not just the low-order six bits.

PROC SUBPROGRAMS

Formal parameters should be called by value whenever possible. If a procedure must reference its formal call by address parameter more than once, a local variable should be declared, set to the value of the formal parameter, and subsequently referenced instead of the formal parameter. Actual call by name parameters are referenced indirectly in the generated code; this level of indirectness can be overcome by evaluating the parameter once and making it local to the PROC (storing the parameter's value in a local variable).

FUNC SUBPROGRAMS

The statements under the heading PROC subprograms are true for FUNC subprograms also. In addition, functions can save two instructions in certain situations. For example: a routine is needed to convert from binary integers to display code, with the result to be stored in one of three arrays, depending upon the section of code where the call originates. If a function is used, as in "ARRAYWORD[I]=FUNCTION[INT];" rather than a procedure, as in "PROCED (INT); ARRAYWORD[I]=INTT;", two SAi k instructions are saved per call. The saving is realized, as functions return their result in register X6 rather than in a core location.

CODING HINTS

Based array references are candidates for scratch variable storage also, if referenced more than once in a sequence of source code, since based array references are indirect.

When storing into many items of the same data structure (array) clustered together, those that refer to the same word of storage should be described in the same order in which they occur.

INDEX

CUT ON THIS LINE

COMMENT SHEET

TITLE: SYMPL Reference Manual Version 1

PUBLICATION NO. 60496400 REVISION A

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

General comments:

FROM NAME: ______________________ POSITION: ______________________

COMPANY NAME: ____

ADDRESS: ____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

FOLD FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

CUT ON THIS LINE

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Publications and Graphics Division
215 Moffett Park Drive
Sunnyvale, California 94086

FOLD FOLD

STAPLE STAPLE

Manual Title SYMPL Version 1 Reference Manual Pub. No. 60496400 Rev. D

As part of Control Data's continuing quality improvement program, we invite you to complete this questionnaire so that you may have a more direct influence on the manuals you use.

Please rate this manual for each general and individual category on a scale of 1 through 5 as follows:

1 - Excellent 2 - Good 3 - Fair 4 - Poor 5 - Unacceptable

I.	Writing Quality	____
A.	Technical accuracy	____
B.	Completeness	____
C.	Audience defined properly	____
D.	Readability	____
E.	Understandability	____
F.	Organization	____
II.	Examples	____
A.	Quantity	____
B.	Placement	____
C.	Applicability	____
D.	Quality	____
E.	Instructiveness	____
III.	Format	____
A.	Type size	____
B.	Page density	____
C.	Art work	____
D.	Legibility	____
E.	Printing/Reproduction	____
IV.	Miscellaneous	____
A.	Index	____
B.	Glossary	____

V. Please provide a yes or no answer regarding manuals in general:

A.	I prefer that a manual on a software product be as comprehensive as possible; physical size is of little importance.	____
B.	I prefer that information on a software product be covered in several small manuals, each covering a certain aspect of the product. Smaller manuals with limited subject matter are easier to work with.	____
C.	I am interested primarily in reference manuals designed for ease of locating specific information.	____
D.	I am interested primarily in user guides designed to teach the user about a product or certain capabilities of a product.	____

VI. We recognize that we have a wide variety of users. Please identify your primary area of interest or activity:

A.	Student	____
B.	Applications programmer	____
C.	Systems programmer	____
D.	How many years programming experience do you have?	____
E.	What languages	
1.	Algol	____
2.	Basic	____
3.	Cobol	____
4.	Compass	____
5.	Fortran	____
6.	PL/I	____
7.	Other	____
F.	Have you ever worked on non-CDC equipment?	____
1.	If yes, approximately what percent of your experience is on non-CDC equipment?	____
2.	How do you rate CDC manuals against other similar manuals using the 1-5 ratings. (Example: XYZ Corp. 2 means XYZ manuals are good as compared to CDC manuals.)	
	Burroughs	____
	DEC	____
	Hewlett-Packard	____
	Honeywell	____
	IBM	____
	NCR	____
	Univac	____
	Other ____________	____

General Comments ____

____

STAPLE

FOLD FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Publications and Graphics Division
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD FOLD

STAPLE STAPLE

60496400

CONTROL DATA
CORPORATION

SYMPL VERSION 1 REFERENCE MANUAL

CDC® OPERATING SYSTEMS:

NOS 1

NOS/BE 1

SCOPE 2

REVISION RECORD

REVISION	DESCRIPTION
A	Original printing.
(11-1-75)	
B	This revision documents SYMPL 1.2, PSR level 439. New features include CONTROL statement
(12-06-76)	additions for trace and optimization. See list of effective pages.
C	This revision documents SYMPL 1.2, PSR level 446. It reflects SYMPL support of the CYBER 170
(03-01-77)	Model 176. See list of effective pages.
D	This revision documents SYMPL 1.3. New features include CONTROL statement
(03-31-78)	addition for weak externals; and points not tested SYMPL control statement option. Appendix F
	contains a glossary.
Publication No. 60496400	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Cover	–
Title Page	–
ii	D
iii/iv	D
v/vi	D
vii, viii	D
1-1, 1-2	B
1-3	C
1-4, 1-5	B
1-6 thru 1-8	D
1-9	B
1-10	C
2-1	B
2-2, 2-3	C
2-4 thru 2-6	D
2-7	C
2-8, 2-9	B
2-10 thru 2-12	D
3-1, 3-2	B
3-3	D
3-4 thru 3-7	B
4-1	B
4-2 thru 4-4	C
4-5	B
4-6 thru 4-8	C
4-9	D
5-1, 5-2	B
5-3	C
5-4	D
5-5	B
5-6, 5-7	C
5-8, 5-9	D
6-1	C
6-2, 6-3	D
6-4	C
6-5, 6-6	D
A-1	A
A-2	D
B-1	D
B-2, B-3	B
B-4	D
C-1	B
C-2, C-3	D
C-4	B
D-1 thru D-3	A
D-4	B
D-5 thru D-7	A
D-8	B
D-9 thru D-15	A
D-16	B
D-17, D-18	A
D-19, D-20	B
D-21	A
D-22	B
D-23 thru D-25	D
E-1, E-2	B
F-1, F-2	D
Index-1 thru -3	D
Comment Sheet	D
Mailer	–
Back Cover	–

Page	Revision

PREFACE

SYMPL version 1.3, which is a systems programming language, operates under control of the following operating systems:

SCOPE 2 for the CONTROL DATA® CYBER 170 Model 176, CYBER 70 Model 76, and 7600 Computer Systems

NOS/BE 1 for the CDC® CYBER 170 Series, CYBER 70 Models 71, 72, 73, 74 and 6000 Series Computer Systems

NOS 1 for the CONTROL DATA CYBER 170 Models 171, 172, 173, 174, 175, CYBER 70 Models 71, 72, 73, 74, and 6000 Series Computer Systems

This reference manual presents the semantics and rules for writing programs in the SYMPL language. It includes sufficient information to prepare, compile, and execute such programs. An appendix presents the syntax of the language in metalinguistic form.

The reader of this manual is assumed to have knowledge of the operating system and computer system under which SYMPL will be used.

Other publications of interest:

Publication	Publication Number
NOS 1 Operating System Reference Manual, Volume 1	60435300
NOS 1 Operating System Reference Manual, Volume 2	60445300
NOS/BE 1 Operating System Reference Manual	60493800
SCOPE 2 Reference Manual	60342600

CDC manuals can be ordered from Control Data Literature and Distribution Services, 8001 East Bloomington Freeway, Minneapolis, MN 55420

CONTENTS

CONSTANTS

SYMPL has five types of constants. Each is a sequence of characters which defines its own value. The constant types are: Boolean, character, integer, real, and status.

BOOLEAN CONSTANTS

Boolean constants represent the two elements of Boolean algebra. They are specified by the reserved words TRUE and FALSE.

CHARACTER CONSTANTS

Character constants represent alphanumeric data. A character constant has the format:

"string"

string	String of 1 through 240 characters of the computer character set shown in appendix A. If the character " is to appear in the string, it must be specified by two consecutive " marks.

For example:

"TAPE01" "ERROR %%"

"QUOTES" "A" " "

INTEGER CONSTANTS

Integer constants represent numeric values. The three types of integer constants are: decimal, octal, and hexadecimal.

During execution, the maximum allowable value for an integer constant depends on the use of the constant. The value of an integer to be converted to a real value and the value of an integer operand for, and the result of, integer multiplication and division must be able to be expressed in 47 bits. High-order bits are lost when a larger value exists, but no diagnostic informs the programmer of such a condition.

Each of the types of integer constants is specified in a different way. Also, each appears in storage in a format appropriate to its type, as described with ITEM declarations for data types.

Decimal Integer Constant

A decimal constant is a string of decimal digits 0 through 9 with an optional preceding + or - sign. The string can contain 1 through 18 digits; it cannot contain blanks. The absolute value for a decimal integer must be able to be expressed in 59 bits.

For example:

+15 -1 4096

Hexadecimal Constant

A hexadecimal constant represents 4 bits in storage for each hexadecimal digit in the constant. The absolute value for a hexadecimal constant must be able to be expressed in 59 bits. If 60 significant bits are written, the leftmost bit is used as a sign in two's complement; and if the constant is stored in a signed integer format of n bits, the nth bit from the right is used as the sign bit.

A hexadecimal constant has the format:

X"string"

string	String of 1 through 15 hexadecimal digits 0 through 9 and A through F. Embedded blanks are ignored.

For example:

X"7FFF" X"9"

Octal Constant

An octal constant represents 3 bits in storage for each octal digit in the constant. If 60 significant bits are written, the leftmost bit is used as a sign in two's complement; and if the constant is stored in a signed integer format of n bits, the nth bit from the right is used as the sign bit.

An octal constant has the format:

O"string"

string	String of 1 through 20 octal digits 0 through 7. Embedded blanks are ignored.

For example:

O"777"   O"33"

REAL CONSTANTS

Real constants represent numeric values in standard single-precision normalized floating point format. A real constant is a string of decimal digits that includes a decimal point and can include a leading sign. Optionally, it can include an exponent representing multiplication by a power of 10. The exponent is specified as either of the semantically equivalent letters D or E followed by an optional plus or minus sign and a decimal integer. A real constant cannot be represented by a string containing an embedded blank.

For example:

3.14E2   -24.   37.E-3

The magnitude limits of a real constant are approximately 10^{-293} to 10^{+322} with up to 15 digits of accuracy. A diagnostic message is given when a number falls outside of the hardware limits.

STATUS FUNCTIONS AND CONSTANTS

Status functions and constants represent small integer values the compiler has associated with the identifiers in a status list. They can be used to preset scalar and array items and can be used in expressions.

Both status constants and status functions require a preceding STATUS declaration to define a status list and identifiers associated with the status list, as described in section 2.

A status function has the format:

stlist"stvalue"

Use of a status function accesses the integer associated with stvalue in status list stlist.

A status constant is a shorthand method of writing a status function. The format of a status constant is:

S"stvalue"

Since a status constant does not indicate which status list it belongs to, it must be used only in a context where the status constant is directly attributable to a particular status list. Such contexts are:

Presetting a scalar or array item of type S.

Joining a status variable by an operator such as:

OPCODE=S"NOP"; IF OPCODE NE S"NOP" . . .

OPERATORS

Operators are used in arithmetic expressions and Boolean expressions. The operators are of type arithmetic, relational, and logical.

Arithmetic operators are of two types:

Numeric operators perform arithmetic operations to yield a numeric result.

Masking operators perform bit-bit-bit operations to yield a numeric result.

Relational operators work with arithmetic operands to produce a Boolean result.

Logical operators work with Boolean values and yield a Boolean result.

Table 1-3 shows the SYMPL symbols (reserved word) and their meanings for the different types of operators. Tables 1-4 and 1-5 show truth tables for the logical and masking operators.

EXPRESSIONS

An expression is a rule for computing a value. During evaluation of an expression the values of the operands in the expression are combined according to the language rules to form a single value.

Each of the following is an expression:

Constant

Scalar

Subscripted array item

Function reference, except the P function

TABLE 1-3. SYMPL OPERATORS

Symbol	Meaning
	Numeric Operators
+	Addition; unary plus.
-	Subtraction; unary minus.
*	Multiplication.
/	Division.
**	Exponentiation.
	Masking Operators
LNO	Logical NOT (bit-by-bit NOT).
LAN	Logical AND (bit-by-bit AND).
LOR	Logical OR (bit-by-bit OR).
LXR	Logical exclusive OR.
LIM	Logical imply.
LQV	Logical equivalent.
	Relational Operators
EQ	Is equal to.
GR	Is greater than.
GQ	Is greater than or equal to.
LQ	Is less than or equal to.
LS	Is less than.
NQ	Is not equal to.
	Logical Operators
NOT	Negation.
AND	Conjunction.
OR	Union.

TABLE 1-4. TRUTH TABLE FOR LOGICAL OPERATORS

b1	False	False	True	True
b2	False	True	False	True
	Logical			
NOT b1	T	T	F	F
b1 AND b2	F	F	F	T
b1 OR b2	F	T	T	T

TABLE 1-5. TRUTH TABLE FOR MASKING OPERATORS

a	0	0	1	1
b	0	1	0	1
	Masking			
LNO a	1	1	0	0
a LAN b	0	0	0	1
a LOR b	0	1	1	1
a LXR b	0	1	1	0
a LIM b	1	1	0	1
a LQV b	1	0	0	1

Further, any of the above entities combined with a unary operator or binary operator also produces an expression.

The two types of expressions are:

Arithmetic expressions that yield numeric values.

Boolean expressions that yield Boolean values of TRUE or FALSE.

Boolean operands and Boolean expressions differ in nature from arithmetic operands and expressions; they cannot be involved with numeric arithmetic expressions. No numeric arithmetic operator applies to any Boolean operand and vice versa.

Evaluation of an expression begins with evaluation of operators with higher precedence and continues with evaluation of operators with lower precedence; otherwise, evaluation proceeds left to right. A different order of evaluation can be specified by the programmer through the use of parentheses: expressions within parentheses are evaluated before the result is combined with other operands.

ARITHMETIC EXPRESSIONS

Arithmetic expressions yield a numeric value. The two types of arithmetic expressions are:

Numeric arithmetic expressions that involve operands of any type except Boolean. Operands are treated as a single value in these expressions.

Logical masking arithmetic expressions that involve operands of any type except Boolean, Operands are treated on a bit-by-bit level in these expressions.

For both types of expressions operators have implicit ranking, with evaluation of the expression preceeding from operators with higher precedence to operators with lower precedence.

Arithmetic operators are as follows. They are listed in order of highest to lowest precedence:

( )	Parentheses, beginning with innermost pair
**	Exponentiation
* /	Multiplication and division, from left to right
+ -	Unary plus and minus
+ -	Addition and subtraction, from left to right
LNO	Logical NOT (complement)
LAN	Logical AND
LOR	Logical inclusive OR
LXR	Logical exclusive OR
LIM	Logical imply
LQV	Logical equivalence

SYMPL has no implicit multiplication in which algebraic multiplication can be indicated by X(Y) or (X)(Y).

Numeric Arithmetic Expressions

A numeric arithmetic expression contains only numeric operands and numeric arithmetic operators. The numeric operators are: **, *, /, +, and -. The numeric operands include constants, scalars, subscripted array items, and function references; the type of any numeric operand must not be Boolean.

When operands of different types are used in a single expression, the compiler converts the type of one operand such that the common type of both operands is the higher type. The four operand types that exist for conversion purposes are as follows, listed in order from highest to lowest:

Real

Signed integer

Unsigned integer

Character.

For example, given integer item I and real item R, the expression (I + R) is evaluated in floating point arithmetic after the value of I is converted to type real. Similarly, the expression ((I + 2) * R) is computed by:

Adding I and 2 in integer mode

Converting the result to floating point format

Multiplying the result by R in floating point format.

Character operands are lowest in the conversion hierarchy. Conversion of type character to type integer is affected by the number of characters declared in the character operand. (The length of a scalar or array item is specified in its declaration; the length of a character constant is the number of characters in the string; the length of a C function is the number of characters indicated in the function.) If bit 59 of a 10 character operand is set, the converted integer is a negative value. If the operand has more than 10 characters, only the first 10 characters are used in an expression evaluation. For operands less than 10 characters, the characters are shifted right to normal integer position and zero filled.

Character-to-real conversion occurs by conversion to integer followed by conversion of the integer to a floating point format.

Conversion from type integer to type real occurs by floating the integer, as provided by hardware instructions. The resulting real value is expressed in single precision format.

Preset VAL to the unsigned integer value 2:

```
STATUS WORDS BEGIN, END, TERM;
ITEM VAL S:WORDS=S"TERM";
```

Set X to 3:

```
STATUS COLOR RED, OR, YEL, BLUE;
X=COLOR"BLUE";
```

Test LETTER for the display code value equivalent to Q:

```
STATUS ALPHA A,B, . . . X,Y,Z;
IF LETTER EQ S"Q" THEN. . .
```

SWITCH DECLARATION

A SWITCH declaration defines a list of label names that the compiler is to associate with small unsigned integer values. The purpose of the declaration is to allow mnemonic references to label names in a GOTO statement.

Two types of switches, and two SWITCH declaration formats, exist. The first is a straightforward list of label names; the second combines STATUS capabilities into the SWITCH declaration.

When a switch is referenced in a GOTO statement, the value of the switch subscript expression must be within the range of defined switches. If the program is compiled with the C parameter (range checking) on the compiler call, an execution-time check is made to determine whether the value is within the range of valid values. When range checking is selected, any value out of range produces a diagnostic and program abort. If range checking is not selected, any reference to an out of range switch value produces an undefined result.

ORDINARY SWITCH

In the simpler form of a switch, the compiler assigns a value to each label named. The first label in the list is assigned a value 0, the second label is assigned the value 1, and so forth.

The format of a SWITCH declaration specifying only label names is:

```
SWITCH swname label, label, . . . ;
```

swname	Name by which switch is known. Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate a reserved word.
label	Label name to be associated with swname. If the switch is never accessed by a particular value, a null parameter (two consecutive commas) can appear in the list for that value.

An example of the declaration and use of an ordinary switch AAA that transfers control to label LAB3 when the value of I is 2 is:

```
SWITCH AAA LAB1, LAB2, LAB3;
GOTO AAA[I];
```

STATUS SWITCH

A status switch references a previously declared STATUS declaration. The SWITCH declaration associates the switch name with a status list; each label name in the switch list is then paired with one of the identifiers from the status list as specified by the SWITCH declaration parameters.

The format of a SWITCH declaration specifying a status list is:

```
SWITCH swname:stlist label:stvalue, label:
   stvalue, . . . ;
```

swname	Name by which switch is known. Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate a reserved word.

stlist	Name by which status list is known, as declared by a previous STATUS declaration.
label	Label name to receive the same value as the status value following the colon.
stvalue	Status value from list stlist to be associated with the preceding label name.

The status values can appear in a switch list in an order other than that of their status list. Also, all of the status values need not be associated with a label. The same label can be associated with more than one status value. A status value, however, can only appear once in a switch list.

An example of a declaration of a status switch WHICHONE and its use to transfer control to LABZ when the value of the GOTO statement argument is 3 is:

```
STATUS COLOR RED, ORG, YEL, GRN;
SWITCH WHICHONE:COLOR LABX:YEL,
   LABZ:GRN;
.
.
.
GOTO WHICHONE[COLOR"GRN"];
```

ARRAY DECLARATION

An ARRAY declaration defines an arrangement of item-like elements. An array can be viewed as a rectangular assortment of entries, each composed of one particular occurrence of each item comprising the entry. The number of entries must be less than 65535.

In storage an array entry occupies an integral number of whole words. Items within the entry can be as small as one bit or as large as 24 words of character data; only type character items can cross the boundary of a word in the array, however.

An array is declared by an ARRAY declaration header followed by an ITEM declaration. If no items exist in the entry, a null declaration (blank followed by a semicolon) should follow the ARRAY declaration. If more than one item (field) exists in the entry, the ITEM declaration should be a compound statement.

The format of an ARRAY declaration header is:

```
ARRAY name [low:up, low:up, . . .]
   alloc (esize),
```

name	Identifier specifying the name of the array. It can be omitted unless the ARRAY declaration appears in a BASED ARRAY, XDEF, or XREF declaration.
low	Lower bound of a dimension of the array, expressed as an integer with modulo 2^{18}. Can be signed positive or negative. If low and its following colon are omitted, 0 is assumed.
up	Upper bound of a dimension of the array, expressed as an integer with a modulo 2^{18}. Can be signed positive or negative. Must be equal or greater than the preceding low with which it is paired.
alloc	Allocation of the entries in the array in storage. P — Parallel allocation in which the first words of each entry are allocated contiguously, followed by the second words of each entry, and so forth. S — Serial allocation in which all the words of one entry are allocated contiguously. If alloc is omitted, P is assumed.
esize	Entry size. Number of words in an array entry, expressed as an unsigned integer. Esize must be less than 2048 words. If esize and its enclosing parentheses are omitted, 1 is assumed.

An array can have up to seven dimensions. Each low:up pair in the ARRAY declaration defines a dimension of the array. (Dimensions specify the coordinates that identify an element of the array.) If the bounds list is omitted, [0:0] is assumed.

Differences between serial and parallel allocation are in figure 2-1. In this figure, array A has one dimension, a three word entry that occurs five times. CHAR[1] is the reference that accesses the second occurrence of item CHAR defined to occupy word 1 of the entry. A full declaration for this array might be:

```
ARRAY A[0:4] S(3);
     BEGIN
     ITEM HDR I(0,0,60);
     ITEM CHAR C(1,0,10);
     ITEM TRFR C(2,0,20);
     END
```

Parallel allocation offers execution advantages and should be used when possible.

The format of the ITEM declaration of an array is as follows. If more than one array item is being declared, all declarations should appear between BEGIN and END. The declaration is similar, but not identical, to the ITEM declaration for scalars.

```
ITEM name type(ep,fbit,size)=[preset],
   name type(ep,fbit,size)=[preset], . . . ;
```

name	Identifier specifying the name of the entry item, expressed as 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate the name of a reserved word. Must be unique within procedure.
type	Type of array item:
	B Boolean
	C Character
	I Signed integer; default
	U Unsigned integer
	R Real
	S:stlist Status associated with list stlist
ep	Entry position. Word number in which the high-order bit of the item occurs, starting from 0; expressed as an unsigned integer constant.
fbit	Bit position at which item begins, starting on the left and counting from 0 through 59; expressed as an unsigned integer constant. For a character item, fbit must be divisible by six.

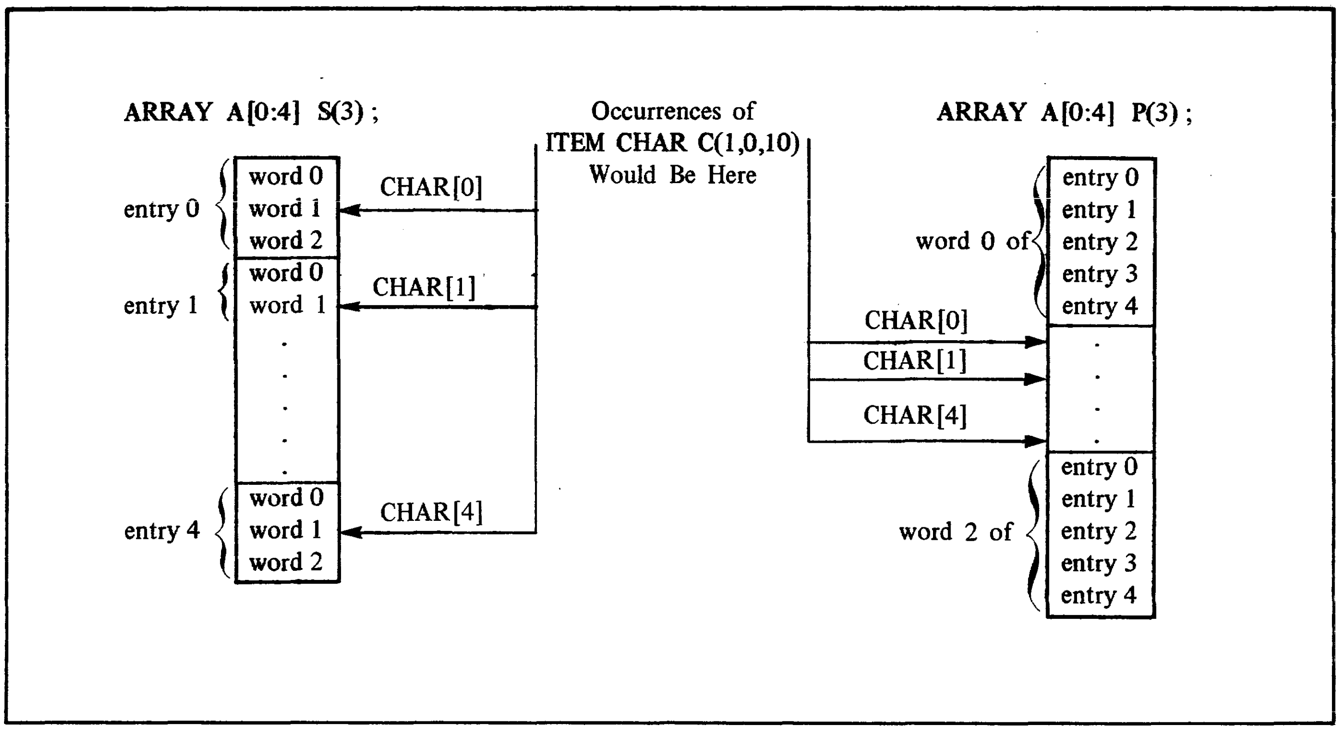


Figure 2-1. Differences in Serial and Parallel Allocation

size	Item length, expressed as an unsigned integer constant appropriate to the type, as shown in table 2-1. Only C type data can cross word boundaries. R type data must have a size of 60.
preset	For a single occurrence array entry item, value to which item is to be initialized at load time, expressed as a constant. For a multiple occurrence array entry item, a set of values arranged in a list in the same order as the allocation order of different instances of the items in storage. Any constant specified is set in the item, aligned appropriately in the field, without regard to other fields in the word.

If the entire field descriptor (ep,fbit,size) is omitted, ep and fbit default to 0 and size defaults as shown in table 2-1. One parameter within the parentheses is assumed to be ep, with fbit=0 and size as in the table; two parameters are assumed to be ep and fbit.

ARRAY REFERENCES

A particular instance of an array item is known as an element. To reference a particular element, a subscript enclosed in brackets is appended to the array item name. For instance:

```
ARRAY REF[0:99];
ITEM REFITEM;
```

To reference the 40th element, which in this example is the 40th word, the reference is:

```
REFITEM[39]
```

The subscript for the array item must be an arithmetic expression. If the type of the arithmetic expression is other than integer, the result of the expression will be converted to integer mode of modulo 2^{17}.

If the array being referenced has more than one dimension, the subscript must have as many arithmetic

TABLE 2-1. ARRAY ITEM DESCRIPTOR LIMITS

Type	fbit Alignment	Maximum Length	Default Length	May Cross Words
I	bit	60 bits	60	no
U	bit	60 bits	60	no
R	bit 0	60 bits	60	no
B	bit	60 bits	1	no
C	byte	240 bytes	1	yes
S	bit	60 bits	60	no

Table 2-1. Array Item Descriptor Limits

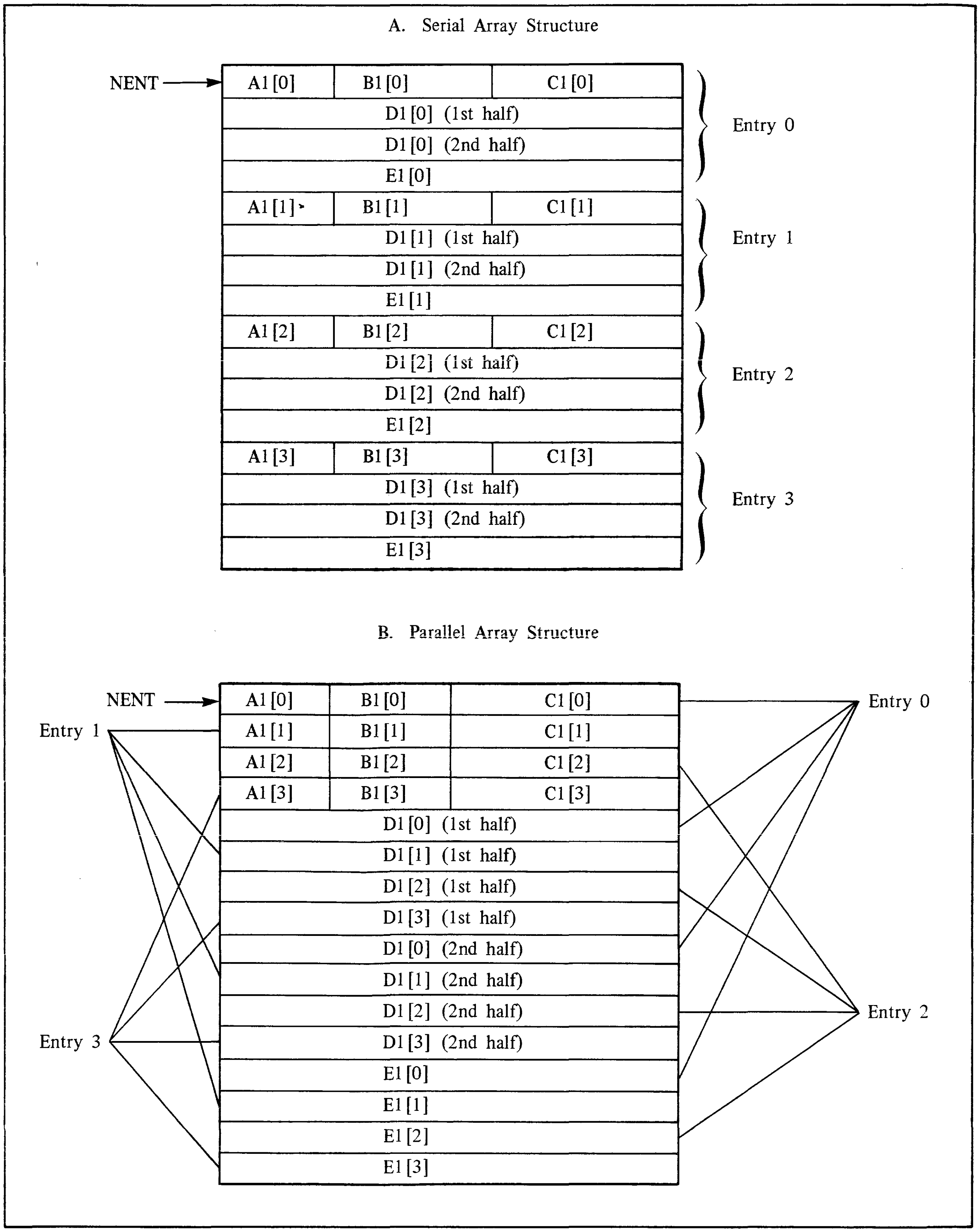
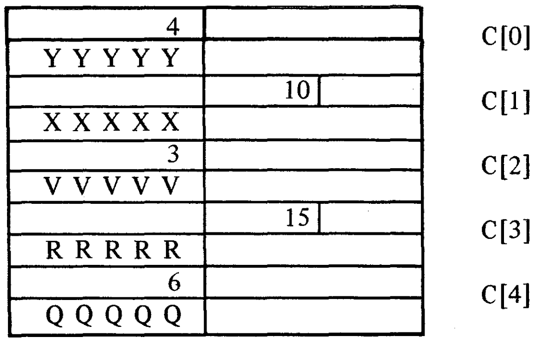


Figure 2-4. Serial and Parallel Arrays with Multiword Items

```
ARRAY TENWORD [0:4] S(2);
BEGIN ITEM A I(0,0,30)=[4, ,3, ,6];
      ITEM B I(0,0,45)=[ , 10, , 15];
      ITEM C C(1,0,5)=["YYYYY","XXXXX",
        "VVVVV","RRRRR","QQQQQ"];
END
```

Resulting structure and values are:



Multidimensional arrays are preset using nested brackets. Brackets should be nested to the level of the number of subscripts. The leftmost subscript varies most rapidly, as it does in FORTRAN Extended.

Basically, the preset list for a declaration is a set of constant values, with the same order as the allocation order of the elements. This list is presented in sections enclosed in square brackets, and nested to a depth of the number of dimensions in the array. An N dimensional array at the first level of nesting has as many sections as the Nth dimension of the array. Each of these sections has as many sections as the N-1st dimension, and so forth. At the deepest level, each section has as many values as the first dimension of the array. Each section at the first level contains values for the instances of the array item with the same rightmost subscript; the subscript associated with each section varying from the lower bound at the left to the upper bound at the right. Each section of the second level contains values for those instances with the same rightmost two subscripts, and so forth. The outermost section is appended to the array item declaration with an equals sign.

Repetition of values can be indicated by bracketing a list of values with a parentheses and a count. For example:

3(2,1)is equivalent to 2,1,2,1,2,1

and

2(2(0,2))is equivalent to 0,2,0,2,0,2,0,2

A two-dimensional parallel array, for example, is initialized by:

```
ARRAY OMEGA[0:1,0:2];
ITEM MU I(0,0)=[[1,2] [3,4] [5,6]];
```

This presetting is equivalent to:

```
ARRAY OMEGA[0:1,0:2];
ITEM MU I(0,0);
MU [0,0]=1;
MU [1,0]=2;
MU [0,1]=3;
MU [1,1]=4;
MU [0,2]=5;
MU [1,2]=6;
```

As with single-dimension arrays, not all elements of a multidimensional array need to initialized. Elements that are not to be initialized can be represented by null brackets as well as by brackets containing null values. For instance:

[[[, ,2] [, 1,]] [[, ,] [3,4,5]] [[, ,] [, ,]]]

is equivalent to

[[[, ,2] [,1]] [[ ] [3,4,5]] [ ]]

Repetition of bracketed sections is indicated by placing a count outside the bracket. For instance:

2[[1,3] [2(2)]]

is equivalent to

[[1,3] [2,2]] [[1,3] [2,2]]

Only the first 6000 words of an array can have preset values.

ARRAY STORAGE AND ADDRESSING

Given the array header:

ARRAY [b_1:u_1, b_2:u_2, . . .] alloc(esize);

the number of entries in the array is:

$$(u_1-b_1+1)(u_2-b_2+1) \ldots (u_n-b_n+1)$$

At compilation time, an array is allocated the following amount of storage:

(number of entries)(esize)

The allocation of an element with respect to the location of its array name is affected by whether storage allocation is serial or parallel.

For serial allocation, the location of element $[s_1,s_2,\ldots,s_n]$ is computed from:

$e_i=s_i-b_i$

$address+ep+e_1(esize)+e_2(size_1+esize)+ \ldots +e_n(size_1* \ldots *size_{n-1}*esize)$

where $size_i$ is u_i-b_i-1 and esize is entry size.

For parallel allocation:

$address+ep*size_1* \ldots *size_{n-1}+e_1+e_2*size_1 + \ldots e_n*size_1* \ldots *size_{n-1}$

where address is the address of element $[b_1, \ldots b_n]$.

For a three-dimension array, the relative location of A[i,j,k] with respect to $A[b_1,b_2,b_3]$ is given by:

location (A[i,j,k])=

location $(A[b_1,b_2,b_3])+(x+L(y+M(z)))$ (esize)

where $x=i-b_1$
$y=k-b_2$
$z=k-b_3$
$L=u_1-b_1+1$
$M=u_2-b_2+1$

A three-dimension array can be initialized, for example, by:

```
ARRAY XYZ[0:2,3:5,-4:-2];
ITEM PI(0,0,60)=[3[3(4)]];
```

Each element of an array resides in a particular row or column. For example:

		column 0	1	2	3
	0	4	0	7	-8
row	1	23	-9	11	6
	2	-7	14	-2	77

In this array, the value 77 resides in row 2, column 3. Because there are three rows and four columns, this array has the dimensions 3 by 4.

Array items are allocated in column order: that is, the leftmost subscript varies most rapidly.

In a two-dimensional array, memory locations are:

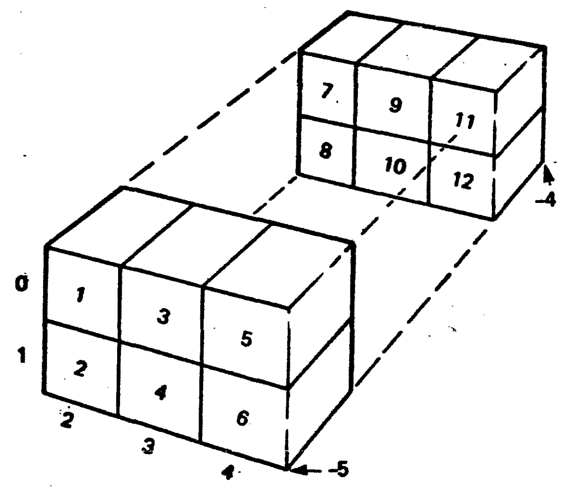
```
ARRAY PSI[1:3,0:3] alloc(2);
ITEM X,Y(1);
```

Parallel	Serial
X[1,0]	X[1,0]
X[2,0]	Y[1,0]
X[3,0]	X[2,0]
X[1,1]	Y[2,0]
X[2,1]	X[3,0]
X[3,1]	Y[3,0]
X[1,2]	X[1,1]
X[2,2]	Y[1,1]
X[3,2]	X[2,1]
X[1,3]	Y[2,1]
X[2,3]	X[3,1]
X[3,3]	Y[3,1]
Y[1,0]	X[1,2]
Y[2,0]	Y[1,2]
Y[3,0]	X[2,2]
Y[1,1]	Y[2,2]
Y[2,1]	X[3,2]
Y[3,1]	Y[3,2]
Y[1,2]	X[1,3]
Y[2,2]	Y[1,3]
Y[3,2]	X[2,3]
Y[1,3]	Y[2,3]
Y[2,3]	X[3,3]
Y[3,3]	Y[3,3]

For a three-dimensional array, the concept and memory locations are:

ARRAY RHO[0:1,2:4,-5:-4] P(1);

Resultant structure of array RHO is shown in figure 2-12.



1	RHO[0,2,-5]
2	RHO[1,2,-5]
3	RHO[0,3,-5]
4	RHO[1,3,-5]
5	RHO[0,4,-5]
6	RHO[1,4,-5]
7	RHO[0,2,-4]
8	RHO[1,2,-4]
9	RHO[0,3,-4]
10	RHO[1,3,-4]
11	RHO[0,4,-4]
12	RHO[1,4,-4]

Figure 2-5. Structure of Array RHO

BASED ARRAY DECLARATION

A based array is an array for which the compiler does not allocate storage; rather the compiler creates a specific pointer variable compiled with an undefined value. All references to a based array are compiled in relation to the pointer variable. From a logical standpoint, a based array provides a template that can be superimposed over any area of memory during execution.

A program using the based array has the responsibility to set the pointer variable through the intrinsic function P. The P function and its use with based arrays is described in section 4.

The based array name is declared in a BASED ARRAY declaration. The array items are declared as they are for normal arrays for which storage is allocated.

The format of the BASED ARRAY header is:

BASED array-dec;

or

BASED BEGIN array-dec, array-dec . . . END

array-dec	Full array declaration including the ARRAY declaration for a header and a simple or compound ITEM declaration for the entry in the array.

Based arrays should be used when the programmer does not know prior to execution time where the array is to be located. Based arrays are used, for instance, with a memory manager such as CMM when the position of an array is not known at load time.

References are made to based arrays just as if they were normal arrays, once the pointer variable is set.

EXCHANGE STATEMENT

The exchange statement causes the exchange of values of the left-hand and right-hand sides of the statement. Appropriate type conversion occurs during the exchange if necessary: in A==B, B is converted as if A=B appeared, with A converted as if B=A appeared.

The format of the exchange statement is:

```
v1 = = v2
```

vi	Entities whose values are to be exchanged. Any of the following can appear: Scalar Subscripted array item P-function Bead function

The two characters == must appear consecutively without an intervening blank.

SYMPL guarantees that subscript or bead function components of expressions which must be evaluated to compute the address of v1 or v2 are computed only once. The order of expansion as to which variable is stored first is not guaranteed, however. The exchange process refers to the expression values by referring to temporary variables. For example, the exchange statement A==B occurs as if it were written:

```
temp=A;
A=B;
B=temp;
```

Temporary variables are used for storage of component and subscript expressions, so that the old values are always used. The expansion of I==J[I] is:

```
temp1=I;
temp2=I;
I=J[I];
J[temp1]=temp2;
```

The subscript expression J[I] is the old value until the statement is complete.

FOR STATEMENT

The FOR statement is a generalized looping control statement. A simple or compound statement following the DO clause of FOR executes repetitively as long as the condition established by the FOR statement is TRUE.

The format of the FOR statement has several forms:

```
FOR i=aexp1 STEP aexp2 DO statement
FOR i=aexp1 STEP aexp2 UNTIL aexp3 DO
statement
FOR i=aexp1 WHILE bexp DO statement
FOR i=aexp1 STEP aexp2 WHILE bexp DO
statement
FOR i=aexp1 DO statement
```

i	Counter for the loop called the induction variable. Must be a scalar of any type except B or C.
aexp1	Arithmetic expression indicating the initial value of the induction variable.
aexp2	Arithmetic expression indicating a value to be added to the induction variable for each execution of the loop.
aexp3	Arithmetic expression indicating the last value for the induction variable for which loop repetition is to occur.
statement	Simple or compound statement to be executed repetitively. This statement is called the controlled statement.
bexp	Boolean expression that must be TRUE for repetitive loop execution.

Since the form FOR i=aexp DO statement produces an infinite loop, the programmer-supplied statement must provide for an exit jump.

The expressions used in the STEP and UNTIL clauses can utilize data of any type. The result of the expression is converted to the mode of the induction variable.

Two types of loops, known as fastloops and slowloops, can be generated by the compiler, depending on the appearance of the compiler-directing CONTROL statement. Figure 3-1 compares the two types of loops.

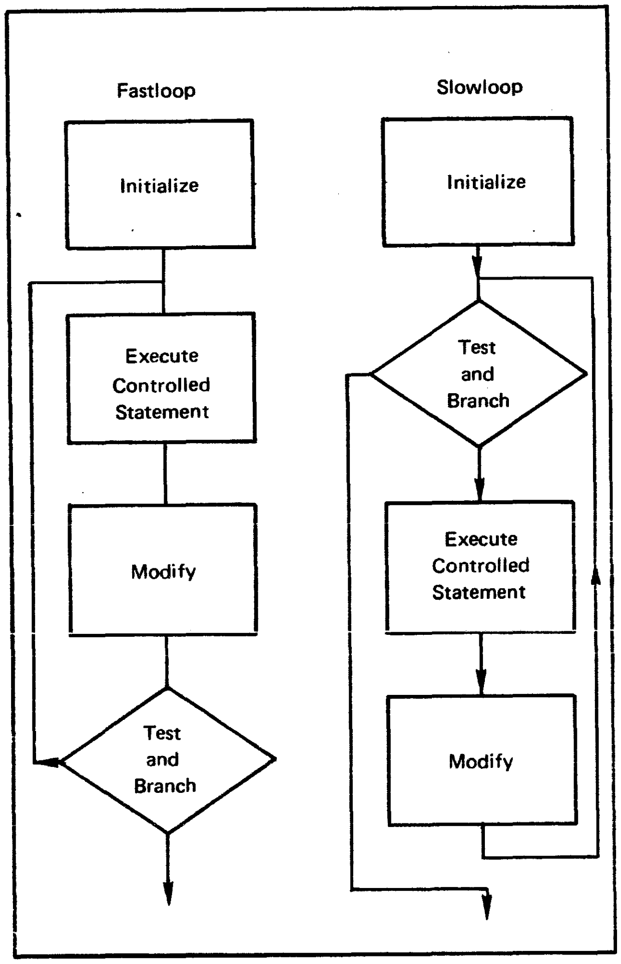


Figure 3-1. Generalized Fastloop and Slowloop Flowcharts

Fastloops always execute at least once (similarly to American National Standard X3.9-1966, FORTRAN DO loops) since the test for the condition is at the end of the loop. To produce predictable results, the elements of the FOR statement are restricted as follows:

> The induction variable must be integer type. It can be signed. The absolute value of the induction variable must be able to be contained within 17 bits.
>
> Neither the step nor the test expression can be modified within the loop. SYMPL might evaluate these expressions before the start of the loop.

Slowloops need not execute at least once since the test for the condition is at the beginning of the loop. The restrictions of fastloops do not hold for slowloops.

Fastloops are preferable since they can be optimized by the compiler.

The default is slowloop, but it can be overridden for following FOR statements: a CONTROL FASTLOOP statement affects all FOR statements begun before a later CONTROL SLOWLOOP statement. A loop control statement within a FOR statement can affect a nested loop, but not the loop in process. See section 5 for an example of loop control.

For both types of loops, the value of the induction variable is undefined after the loop is complete. For slowloops, however, the current value of the induction variable is preserved if the controlled statement causes a jump out of the loop. Moreover, if the controlled statement is entered by a GOTO statement from outside the FOR statement, the value of the induction variable might be undefined.

Figure 3-2 shows the different types of FOR statements and the logic of their generated code. For slowloops, the object code has a direct correspondence with the SYMPL statements shown; this is not the case with fastloops.

The step value and final value shown in figure 3-2 in temporary locations are not guaranteed: if variables involved in these expressions are modified within the loop, results are not predictable.

TEST STATEMENT WITHIN A FOR STATEMENT

In a FOR statement, the compiler automatically supplies the modification, test, and branching steps of a loop. The TEST statement provides a means of branching to the modify-test-branch step; it is meaningful only within the controlled statement of a FOR statement.

The format of the XDEF declaration is:

```
XDEF xdec
or
XDEF BEGIN xdec xdec . . . END
```

xdec	Name of any procedure, function or label that is to be referenced in an externally compiled program; or a full data declaration for a scalar, array, switch, or based array.
	The xdec for a procedure, function or label is:
	PROC name;
	FUNC name type;
	LABEL name, name, . . . ;

XDEF declarations for procedure and function names can occur either before or after the declarations of the procedure or function.

An example of use of the XDEF and XREF declarations is as follows:

Procedure A is compiled with:

```
XREF ITEM COUNT I;
```

Procedure B is compiled with:

```
XDEF ITEM COUNT I;
```

Any reference to COUNT from within procedure A accesses the storage reserved for the item within procedure B, assuming both A and B are available at load time.

XREF DECLARATION

The XREF declaration generates external references to the specified names. It is assumed that storage for variables is allocated and appropriately declared with XDEF in a separately compiled program.

The format of the XREF declaration is:

```
XREF xdec
or
XREF BEGIN xdec xdec . . . END
```

xdec	Any of the following whose storage is declared with XDEF:
	Data declaration for a scalar without preset.
	Data declaration for an array without presets.
	Data declaration for a based array.
	PROC name;
	FUNC name type;
	LABEL name, name, . . . ;
	SWITCH name, name, . . . ;

XREF itself is not terminated by a semicolon, but each declaration within the XREF statement requires a terminating semicolon.

Examples of XREF statements are:

```
XREF BASED ARRAY AA; ITEM XX;

XREF BEGIN
     SWITCH JUMVEC;
     FUNC LINEUP R;
     ARRAY[0:9,0:9] S(5);
        BEGIN
        ITEM ZZ C(0,0,40);
        ITEM YY R(4,0,60);
        END
     END
```

Each parameter in the actual parameter list is delimited by the final parenthesis or a comma. A parameter consists of all the characters between successive parameter delimiters.

Any character can appear as part of the actual parameter string, but characters with syntax-defining meaning might require special coding:

> Any parameter string that contains a semicolon must be bounded by #. The bounding # are removed prior to substitution.
>
> Any parameter string that contains # must specify ## to produce a single # substitution.
>
> Any parameter string that contains incorrectly unbalanced or nested ( ), < > , or [ ] must be bounded by #. The bounding # are removed prior to substitution.
>
> Any comma within a parameter string is not recognized as a parameter delimiter when that comma is contained within a balanced set of ( ), < >, or [ ].

All actual parameters valid for a procedure or function call are valid as DEF parameter strings. No restriction limits the DEF name reference parameter strings to items or expressions, however.

For example:

- Define BYTE and reference it by BYTE(C,5,2**J):

  ```
  DEF BYTE(B,J,K) # B<J>A[K] #;
  ```

  Expansion produces:

  ```
  C<5>A[2**J]
  ```

- Define CHECK with two parameters and a body that uses the BYTE specified above:

  ```
  DEF CHECK(X,ERROR) # IF BYTE(B,1,X)
     EQ 1 THEN GOTO OK; ERROR#;
  ```

  Reference:

  ```
  CHECK(CALL(3,B),#ERROR=37;
     GOTO FAIL#);
  ```

  Expansion:

  ```
  IF B<1>A[CALL(3,B)] EQ 1 THEN GOTO
     OK; ERROR=37; GOTO FAIL;
  ```

- Another definition of CHECK with the same parameters produces the following expansion, given the same reference:

  ```
  DEF CHECK(X,ERROR)#IF BYTE
     (B,1,##X##) EQ 1 THEN GOTO OK;
     ERROR#;
  ```

  Expansion:

  ```
  IF B<1>A[X] EQ 1 THEN GOTO OK;
     ERROR=37; GOTO FAIL;
  ```

DEF NAME REFERENCES

Once a DEF name has been defined, subsequent references to that name are replaced by the characters in DEF body. No substitution occurs in the following circumstances, however:

> The DEF name appears within a comment.
>
> The DEF name appears within a constant.
>
> The DEF name or the DEF parameter name appears as the identifier being defined by an ITEM, ARRAY or COMMON declaration.
>
> The DEF name corresponds to one of the following and the name appears in a syntax-defining context:
>
> > Type descriptor abbreviations B, C, I, R, S, U.
> >
> > Array layout specifiers P, S.
> >
> > Constant prefixes O, S, X.
> >
> > Intrinsic function B, C, P.
> >
> > Real number specifiers D, E.

When the DEF declaration does not include parameters, compilation simply replaces the DEF name with the DEF body.

When the DEF declaration includes parameters, each reference to the DEF name must be followed by an actual parameter list. The format of the DEF name reference with parameters is:

```
name(param,param, . . . )
```

name	Name defined in a prior DEF declaration within this subprogram.

param	String of characters to replace a formal parameter.

No comment can appear between the DEF name and the left parenthesis of the actual parameter list.

A one-to-one correspondence exists between the positions of parameters in each list. The first actual parameter replaces all occurrences of the first formal parameter within the DEF body; the second actual parameter replaces all occurrences of the second parameter; and so forth. The number of actual parameters must not exceed the number of formal parameters: such a condition is detected as a fatal error and DEF name substitution is suppressed.

The number of actual parameters can be fewer than the number of formal parameters, however. Any formal parameter without a corresponding actual parameter is replaced by a null character string. This allows the expansion of a DEF name with a variable number of actual parameters.

CONTROL STATEMENT

The CONTROL statement directs the compiler to take immediate action. Several different types of control words in the statement cause different types of actions:

Output listing control specifications are EJECT, LIST, NOLIST, OBJLST.

Conditional compilation control words are IF, FI, ENDIF.

Compilation option selections are PACK, PRESET, FTNCALL.

FOR statement loop specifications are FASTLOOP, SLOWLOOP.

Core residence selections are LEVEL1, LEVEL2, LEVEL3.

Variable attribute specifications are DISJOINT, OVERLAP, REACTIVE, INERT.

Weak external specification is WEAK.

Traceback selection is TRACEBACK.

Each of the different functions is described separately below.

A CONTROL statement can appear anywhere in a program that a statement can appear. It can also appear within BEGIN and END enclosing a list of array items, based arrays, external declarations, or common declarations.

The effect of a CONTROL statement can be reflected in an entire module. The end of a procedure or function does not cancel the statement; only TERM cancels a CONTROL statement.

LISTING CONTROL

Four forms of the CONTROL statement affect output listings. The general format is:

CONTROL control-word;

Control-word	One of the following:
EJECT	Skip to new page of listing
LIST	Resume normal listing of source statements
NOLIST	Suspend normal listing of source statements
OBJLST	List object code

EJECT, LIST, and NOLIST cause the compiler to take action at the time the statement is encountered among the source statements.

OBJLST applies to the entire module. Its appearance anywhere within the module affects the entire module.

The H parameter of the SYMPL compiler call overrides CONTROL NOLIST.

CONDITIONAL COMPILATION

The CONTROL statement can be used to determine whether source statements following the CONTROL statement are to be compiled:

When the relationship defined in the CONTROL statement tests TRUE, the following source statements are compiled.

ATTRIBUTES OF VARIABLES SPECIFICATION

The SYMPL compiler attempts to produce efficient executable code. Because the compiler cannot predict the precise use of a variable in subsequent source statements, it must forego many efficiencies that would produce inaccurate code by particular variable references. The programmer, however, can be aware of data use and, through the CONTROL statement, can inform the compiler of usage characteristics. By classifying variables and array items as separate or potentially overlapping, the programmer provides the information that the compiler needs to decide optimizations.

The format of the CONTROL statement for specifying attributes of variables is:

```
CONTROL attribute var, var, . . . ;
or
CONTROL attribute;
```

attribute — Attribute of variables in the statement list:

OVERLAP	Variables might be referenced by more than one name, as shown in examples below. OVERLAP is the opposite of DISJOINT.
DISJOINT	Variables are referenced by a single name only. DISJOINT is the opposite of OVERLAP.
REACTIVE	A given word in a single array might contain two items, or parts of items, being referenced together although the two items are not declared to overlay each other. See examples below. REACTIVE is the opposite of INERT. Items with declarations that show one field overlaying another field are detected by the compiler, so that REACTIVE need not be declared.
INERT	A given word in a single array does not contain items, or parts of items, referenced together. INERT is the opposite of REACTIVE.

var — Variable with the attribute specified.

If the list of variables is omitted, the CONTROL statement becomes a global switch that affects all subsequently declared variables not otherwise referenced by a contrary individual specification.

If neither the global switch format nor the individual specification format of the CONTROL statement appears, the module is compiled as described in appendix C, Possible Optimizations. If any CONTROL statement specifying an attribute appears in the module, the global switch format CONTROL REACTIVE and CONTROL OVERLAP is assumed at the beginning of a module. Use of the CONTROL statement to classify variables is encouraged because future versions of the compiler might require such classification.

The definitions of overlap and disjoint refer only to variables in separate arrays; for overlapping items within a single array, the distinction between reactive and inert must instead be drawn.

Overlapped Variables

One program might refer to the same variable by two names when formal parameters or based arrays are referenced. For example:

```
PROC P(A,B);
.
.
.
A=2;
B=4;
Y=A;
```

A call to procedure P in the form P(V,V) represents two occurrences of the same actual parameter: during compiler optimization the store of the value of Y must not use the value of A from the A=2 statement.

Similarly, with a based array B based on A:

```
PROC P(A,B);
X=A[2];
B[2]=3;
Y=A[2];
```

Since A and B refer to the same array, the compiler must not store Y such that it refers back to the first A[I].

Variable names that interfere with each other as illustrated above are called overlapped variable names. If such interference does not occur, the variables are said to be disjoint.

To determine whether variables should be specified as OVERLAP or DISJOINT, the programmer must examine the entire module, not simply a given subprogram. The compiler reserves the right to inspect all procedures and functions in a given module for use of variables and it considers that normal nonexternal variables are not destroyed by calls to global subprograms whether external or not. But if local procedures are called which have access to the names of local variables, the compiler detects all the variables such a procedure explicitly stores.

Variables known through COMMON, XDEF, and XREF declarations are considered destroyed by calls to an external subprogram. Overlapped behavior exists when an external subprogram destroys nonexternal variables.

Reactive Arrays

Two items in one array can interfere with optimization when references to items do not match the declarations of these items. For example:

```
ARRAY [0:100] S (1);
  ITEM A (0), B (1);
.
.
.
B[I]=A[J]*2;
Q=A[J];
```

Item B is outside the bounds of one array entry and it interfers with the next entry. If the array is always indexed by 2, B does not interfere with A. However, if I is set to J-1, the A(J) is destroyed by a store to B(I).

Array items that interfere with each other as in this illustration are said to be reactive items. If such interference does not occur, the items are said to be inert. An array is reactive if it has two items A and B such that for A[i] and B[j] with i not equal to j at some time during execution, any part of A[i] is in the same word as any part of B[j]. It is not necessary for the fields to overlap: reactive arrays occur when both items are in the same word.

To determine whether an array item should be classified as REACTIVE or INERT, the programmer must examine an entire module, including all variables affected by other procedures it might call.

WEAK EXTERNALS

When a compiled program is loaded before execution, the loader searches for a matching entry point for all externals and loads the subprogram in which they occur. Under some circumstances this can result in the loading of subprograms not required for current execution. Through using a CONTROL statement to declare an external weak, the programmer can specify that the external is not necessarily to be satisfied.

A weak external does not cause a search for the matching entry point. If the program that contains the entry point is loaded for some other reason, however, that weak external is linked.

When a weak external is satisfied, it is linked as if it were a normal external. If it is not satisifed, no error message is produced.

The format of the CONTROL statement specifying a weak external is:

```
CONTROL WEAK name, name, . . . ;
```

name	Name of array, based array, function, item, label, procedure, or switch. Name must have been previously declared as external by using XREF.

TRACEBACK FACILITY

SYMPL uses standard calling sequences for transferring control to a procedure or subroutine of another language. In this sequence, register A1 contains the address of a parameter list and each parameter to be passed occupies one word of the list. Execution of an RJ instruction to the entry point links the programs. For debugging purposes, SYMPL provides an option for traceback.

The format of the CONTROL statement for tracing purposes is:

```
    CONTROL TRACEBACK;
```

The appearance of this statement anywhere within the module selects the option for the entire module. Traceback code is generated automatically when the K parameter (points-not-tested) of the SYMPL compiler call is used.

The traceback code generated for procedures and functions is compatible with traceback of FORTRAN Extended. To complete FORTRAN Extended compatibility, the F parameter of the SYMPL compiler call must also be specified. Code generated by a SYMPL calling program is never compatible with FORTRAN Extended traceback, however.

Traceback code generated is as follows:

If the procedure of function has a single entry, the generated constant word is:

```
VFD 42/0Hname,18/ept
```

name	Subprogram name left-justified and blank filled or truncated to seven characters.
ept	Address of subprogram entry point.

If the procedure or function has multiple entries, the generated constant word is:

```
VFD 42/0Hname,18/temp
```

name	Subprogram primary entry point.
temp	Address of a copy of the return information taken from the most recent entry point.

The return jump instruction for the subprogram call is forced upper. The lower 30 bits of the instruction contain:

```
VFD 12/line,18/trace
```

line	Approximate source line number of call.
trace	Address of the constant word described above for the innermost subprogram containing the call statement.

COMPILER CALL AND OUTPUT LISTINGS 6

COMPILER CALL

The SYMPL compiler is called with a control statement that conforms to operating system syntax. The control statement cannot be continued.

More than one program or subprogram can be compiled by a single call to the compiler as long as they follow each other on the source file without any file boundaries between them. The compiler recognizes a TERM statement as the end of a module and ignores any further statements on the same card or card image. Compilation resumes with the next card, which is assumed to be the start of another program or subprogram. A comment can precede a program or subprogram header.

If the first card or card image encountered at the beginning of a loader module contains the character OVERLAY in columns 1 through 7, the remainder of the module is treated as if an LCC statement appeared in a COMPASS program.

The name on the compiler call statement is SYMPL. If all default parameters are selected, the compiler call appears as:

```
SYMPL.
```

A variety of compilation options can be specified in a parameter list following the compiler call name. If the name of the source input file is NEWONE, for example, the compiler call appears as:

```
SYMPL,I=NEWONE.
```

All compilation parameters are optional and can appear in any order. Parameters are listed below in alphabetical order.

A ABORT JOB AFTER ERRORS

omitted	Execute next control statement whether or not any errors are diagnosed during compilation.
A	Execute control statement after an EXIT(S) control statement if errors are found at the end of compilation.

B BINARY CODE FILE

omitted	Write binary output from compilation to file LGO.
B	Write binary output from compilation to file LGO.
B=0	Suppress generation of binary code.
B=lfn	Write binary output from compilation to file lfn, where lfn is one through seven letters or digits beginning with a letter.

C CHECK SWITCH RANGE

omitted	Do not generate code to check range of switch references. Any reference to an undefined switch value produces either an endless loop, a mode error, or a wild jump.
C	Generate code to check range of switch references. During execution any reference to an out-of-range switch or an unspecified switch value produces a diagnostic and a program abort.

D PACK SWITCHES

omitted	Generate one word for each switch.
D	Generate one word with two switch points, reducing the size of generated code but increasing execution time. Produces the same result as CONTROL PACK within a program.

E COMPILE $BEGIN/$END STATEMENTS

omitted	Do not compile source statements bracketed between $BEGIN and $END.
E	Compile source statements bracketed between $BEGIN and $END.

F FORTRAN CALLING SEQUENCE

omitted	Do not compile a word of all zeros at the end of a parameter list.
F	Compile a word of all zeros at the end of each parameter list as required by the FORTRAN Extended calling sequence. Produces the same result as a CONTROL FTNCALL statement within a program.

H LIST ALL SOURCE STATEMENTS

omitted	List source statements according to CONTROL NOLIST and CONTROL LIST statements within the program.
H	List all source statements, regardless of CONTROL NOLIST statements within the program.

I SOURCE INPUT FILE

omitted	Compile card images from file INPUT.
I	Compile card images from file COMPILE.
I=lfn	Compile card images from file lfn.

K POINTS-NOT-TESTED

omitted	Do not generate points-not-tested interface code.
K	Generate an RJ to the points-not-tested interface routine after every label and conditional jump. Find all paths in the executable code and determine which of the paths are exercised by the test base. Also, generate traceback code.

L LISTING FILE

Any O, R, or X parameter must be concatenated with any L parameter, as in: LXOR=PRINTIT.

omitted	Write source statement listing and diagnostics to file OUTPUT.
L	Write source statement listing and diagnostics to file OUTPUT.
L=1	Write summary of resources used to file OUTPUT.
L=0	Suppress all listing output, including that selected by O, R, and X; list only diagnostics.
L=lfn	Write source statement listing and diagnostics to file lfn, with lfn being one through seven letters or digits beginning with a letter.

N CROSS REFERENCE UNREFERENCED ITEMS

omitted	List only referenced items on the cross reference map selected by the R parameter.
N	List referenced and unreferenced data items on the cross reference map selected by the R parameter.

O LIST OBJECT CODE

Any L, R, or X parameter must be concatenated with any O parameter, as in: OL=LIST/35/45.

omitted	Do not list binary object code.
O=st/end	List binary object code generated by range of source statements indicated: st Number of first source statement whose object code is to be listed. Default is 0. end Number of last source statement whose object code is to be listed. Default is last statement in program. If only one number appears after =, it is presumed to be end. The line numbers appear to the left of the source images on the listing.
O=lfn/st/end	List binary object code from specified source statements on file lfn, where lfn is one through seven letters or digits beginning with a letter. st and end are as above.

P PRESET COMMON

omitted	Data items in common blocks are not to be initialized.

P	Initialize data items in common blocks according to the preset values in the data declarations. Produces the same result as a CONTROL PRESET statement within a program.

R LIST CROSS-REFERENCE MAP

Any L, O, or X parameter must be concatenated with any R parameter, as in: RX=SHOW.

omitted	Do not list cross reference table and common blocks.
R	List cross reference table and common blocks on file OUTPUT.
R=lfn	List cross reference table and common blocks on file lfn, where lfn is one through seven letters or digits beginning with a letter.

S EXECUTION LIBRARY

omitted	Compile LDSET tables with references to these libraries: SYMLIB/FORTRAN for NOS and NOS/BE operating systems SYMIO/FORTRAN for SCOPE 2 operating system
S=0	Suppress LDSET table generation.
S=lib	Generate LDSET tables with references to library lib. Multiple libraries can be specified with slashes between library names, as in: S=AAA/MMM/TTT.

T SYNTAX CHECK

omitted	Check syntax and generate binary code.
T	Check syntax, but do not generate binary code.

W SINGLE STATEMENT CODE GENERATION

omitted	Generate object code with multiple source statement intermixed.
W	Generate object code that maintains a close correspondence with its source statement. While the resulting object code might be less efficient, it is useful for debugging.

X LIST STORAGE MAP

Any L, R, or O parameter must be concatenated with any X parameter, as in: RX=OUTPUT.

omitted	Do not list storage map or common blocks.
X	List storage map and common blocks on file OUTPUT.
X=lfn	List storage map and common blocks on file lfn, where lfn is one through seven letters or digits beginning with a letter.

Y SUPPRESS DIAGNOSTIC 136

omitted	List diagnostic 136 (Semi ends comment) as required.
Y	Suppress diagnostic 136 listing, but take normal corrective action.

OUTPUT LISTINGS

Figure 6-1 shows a SYMPL main program SORT100 that can be used to sort 100 items. It calls procedure SORTER which was compiled separate from SORT100 since TERM appeared at the end of SORT100. SORT-100 consequently contains an XREF statement that declared SORTER to be an external program.

A job deck for syntax analysis compilation both the main program and subprogram would appear as:

```
jobcard.
any accounting statement.
SYMPL,T.
7/8/9
all SYMPL source statement
6/7/8/9
```

Output from a compilation normally includes the source statement listing, and a diagnostic summary.

```
PRGM SORT100  ;
BEGIN
BASED ARRAY AA[99]  ;
      ITEM X  ;
ITEM NOREFERENC;
XREF PROC SORTER;
ARRAY TOBESORTED [99]  ;
      ITEM T;
P<AA> = LOC(TOBESORTED)   ;
SORTER (P<AA>) ;
END
TERM
      PROC SORTER(SORT) ;
      BEGIN
      ARRAY SORT[99];
          ITEM VALUE;
      ITEM I;
      ITEM FLAG I=0  ;
L1:   FOR I=0 STEP 1 UNTIL 98 DO
                IF VALUE[I+1] GR VALUE[I] THEN
                BEGIN
                VALUE[I+1] == VALUE[I];
                FLAG = 1;
                END
      IF FLAG EQ 0 THEN
               RETURN;
      FLAG = 0 ;
      GOTO L1;
      END =SORTER=
      TERM
```

Figure 6-1. Sample Source Program

Any storage map, cross-reference map, or object listing follows on a separate page of the listing. The last information shown summarizes the number of words of memory and the time required for compilation. The parameters of the compiler call used for compilation, whether selected explicitly or implicitly, are also shown.

A large map might appear on the output listing in two parts. Both should be examined.

STORAGE MAP

The storage map is a dictionary of all programmer-created declarations in the source program. It is selected by the X parameter of the compiler call. Figure 6-2 shows the storage map from the SORT100 main program of figure 6-1. Information appearing on the map includes:

1 NAME First ten characters only of declarations are printed.

2 TYPE Defines the name as one of the following types:

ARYITM	Array item
COMMON	Common block
ITEM	Item
FUNC	Function
PROC	Procedure
LABEL	Label
B.ARRY	Based array
ARRAY	Array
PROGRAM	Program

3 M Mode of data representation

B	Boolean
C	Character
I	Integer
P	Parallel (arrays only)
S	Status (Serial if type is array or based array)
U	Unsigned integer
X	External
Y	Weak external

4 LOC Octal address relative to start of routine; if followed by C, LOC is relative to start of common block. If type = ARYITM, LOC refers to first occurrence of item.

5 FBIT First bit, numbered from 0 to 59, left to right.

6 NUM Number of bits; if MODE = C, number of bytes.

CROSS-REFERENCE MAP

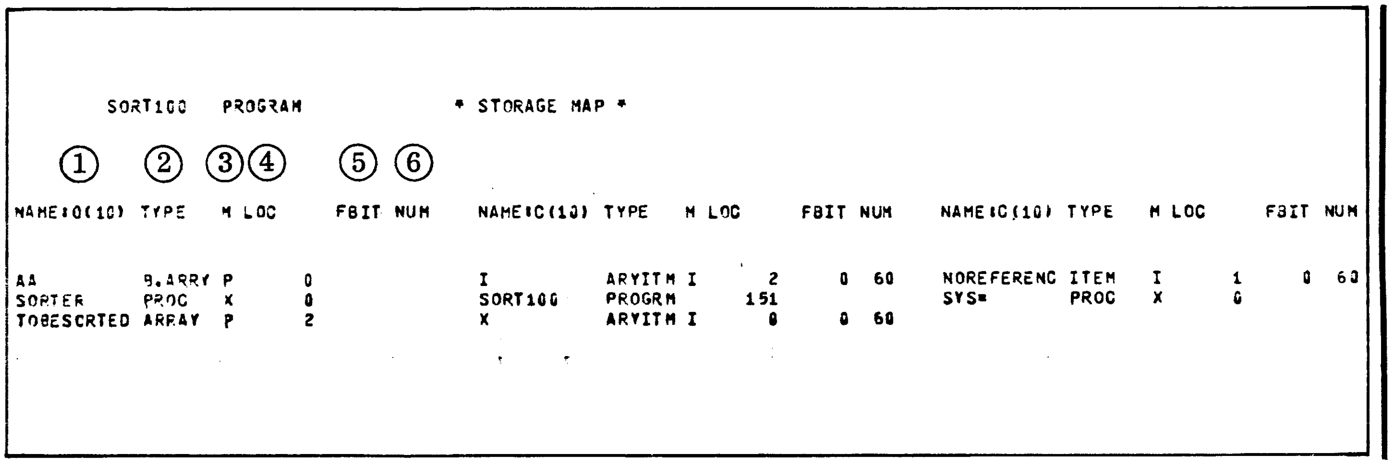
The cross-reference map lists the properties of each declaration and shows the source line number at which the entity was declared or referenced. It is selected by the R parameter of the compiler call.

Figure 6-3 shows the cross-reference map from subprogram SORT100 of figure 6-1. Since the subprogram was compiled with the N parameter of the SYMPL compiler call, items that were declared, but not referenced, also appear on the map. Information appearing on the map includes:

1 NAME First ten characters only of declarations are printed.

2 TYPE Defines the name as one of the following types:

ARYITM	Array item
COMMON	Common block
ITEM	Item
FUNC	Function
PROC	Procedure
LABEL	Label
B.ARRY	Based array
STSCON	Status constant
DEFINE	DEF
STSLST	Status list
PROGRM	Program
ARRAY	Array



```
     SORT100   PROGRAM                * STORAGE MAP *

   ①     ②   ③④      ⑤  ⑥

NAME:O(10) TYPE   M LOC     FBIT NUM    NAME:C(10) TYPE   M LOC      FBIT NUM    NAME:C(10) TYPE   M LOC      FBIT NUM

AA         B.ARRY P     0               I          ARYITM I     2      0  60    NOREFERENC ITEM   I     1      0  60
SORTER     PROC   X     0               SORT100    PROGRM     151                SYS=       PROC   X     0
TOBESCRTED ARRAY  P     2               X          ARYITM I     0      0  60
```

Figure 6-2. Storage Map

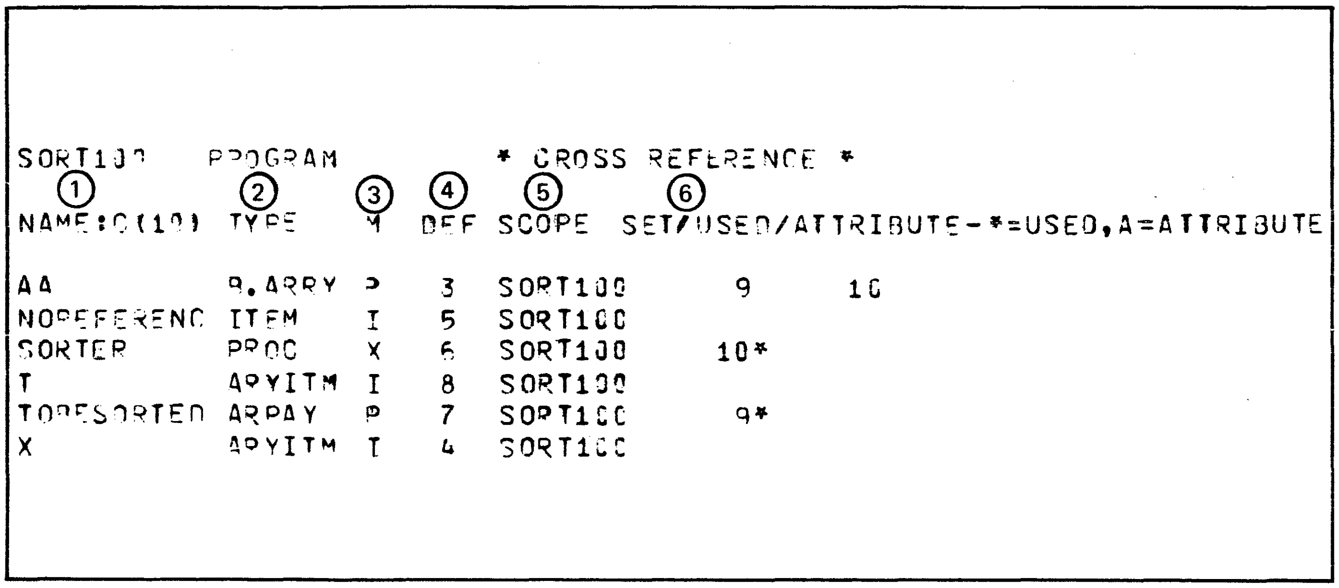
3 M Mode of data representation

B	Boolean
C	Character
I	Integer
P	Parallel (arrays only)
S	Status (serial if type = array)
U	Unsigned integer
X	External
Y	Weak external

4 DEF Line number in source listing where declaration is defined; if followed by C, declaration is in common block.

5 SCOPE Name of outermost procedure within which declaration occurs; if type = STSCON, SCOPE is the name of the status list of which the item is a member.

6 SET/USED Source listing line numbers of references to NAME, * indicates use as other than left-hand side of the replacement statement.



```
SORT100     PROGRAM               * CROSS REFERENCE *
   ①          ②        ③    ④    ⑤       ⑥
NAME:C(10)  TYPE       M   DEF  SCOPE  SET/USED/ATTRIBUTE-*=USED,A=ATTRIBUTE

AA          R.ARRY     P    3   SORT100       9      10
NOREFERENC  ITEM       I    5   SORT100
SORTER      PROC       X    6   SORT100      10*
T           ARYITM     I    8   SORT100
TOBESORTED  ARRAY      P    7   SORT100       9*
X           ARYITM     I    4   SORT100
```

Figure 6-3. Cross-Reference Map

STANDARD CHARACTER SETS A

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation was specified when the operating system was installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multi-punched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of the standard character set table are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

Display Code (octal)	CDC			ASCII		
	Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
00[†]	: (colon)[††]	8-2	00	: (colon)[††]	8-2	072
01	A	12-1	61	A	12-1	101
02	B	12-2	62	B	12-2	102
03	C	12-3	63	C	12-3	103
04	D	12-4	64	D	12-4	104
05	E	12-5	65	E	12-5	105
06	F	12-6	66	F	12-6	106
07	G	12-7	67	G	12-7	107
10	H	12-8	70	H	12-8	110
11	I	12-9	71	I	12-9	111
12	J	11-1	41	J	11-1	112
13	K	11-2	42	K	11-2	113
14	L	11-3	43	L	11-3	114
15	M	11-4	44	M	11-4	115
16	N	11-5	45	N	11-5	116
17	O	11-6	46	O	11-6	117
20	P	11-7	47	P	11-7	120
21	Q	11-8	50	Q	11-8	121
22	R	11-9	51	R	11-9	122
23	S	0-2	22	S	0-2	123
24	T	0-3	23	T	0-3	124
25	U	0-4	24	U	0-4	125
26	V	0-5	25	V	0-5	126
27	W	0-6	26	W	0-6	127
30	X	0-7	27	X	0-7	130
31	Y	0-8	30	Y	0-8	131
32	Z	0-9	31	Z	0-9	132
33	0	0	12	0	0	060
34	1	1	01	1	1	061
35	2	2	02	2	2	062
36	3	3	03	3	3	063
37	4	4	04	4	4	064
40	5	5	05	5	5	065
41	6	6	06	6	6	066
42	7	7	07	7	7	067
43	8	8	10	8	8	070
44	9	9	11	9	9	071
45	+	12	60	+	12-8-6	053
46	-	11	40	-	11	055
47	*	11-8-4	54	*	11-8-4	052
50	/	0-1	21	/	0-1	057
51	(	0-8-4	34	(	12-8-5	050
52	)	12-8-4	74	)	11-8-5	051
53	$	11-8-3	53	$	11-8-3	044
54	=	8-3	13	=	8-6	075
55	blank	no punch	20	blank	no punch	040
56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
57	. (period)	12-8-3	73	. (period)	12-8-3	056
60	≡	0-8-6	36	#	8-3	043
61	[	8-7	17	[	12-8-2	133
62	]	0-8-2	32	]	11-8-2	135
63	%[††]	8-6	16	%[††]	0-8-4	045
64	≠	8-4	14	" (quote)	8-7	042
65	↱	0-8-5	35	_ (underline)	0-8-5	137
66	∨	11-0 or 11-8-2[†††]	52	!	12-8-7 or 11-0[†††]	041
67	∧	0-8-7	37	&	12	046
70	↑	11-8-5	55	' (apostrophe)	8-5	047
71	↓	11-8-6	56	?	0-8-7	077
72	<	12-0 or 12-8-2[†††]	72	<	12-8-4 or 12-0[†††]	074
73	>	11-8-7	57	>	0-8-6	076
74	≤	8-5	15	@	8-4	100
75	≥	12-8-5	75	\	0-8-2	134
76	¬	12-8-6	76	¬ (circumflex)	11-8-7	136
77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

[†]Twelve zero bits at the end of a 60-bit word in a zero byte record are an end of record mark rather than two colons.

[††]In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_8).

[†††]The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

DIAGNOSTICS B

The SYMPL compiler recognizes errors in SYMPL syntax. An applicable diagnostic message is printed on OUTPUT immediately preceding the line on which the error was detected. In addition, the total number of diagnostic messages is printed along with a detailed listing of each message number and the condition that caused the error.

The compiler aborts under several conditions:

Error in the compiler call. A dayfile message PARAMETER n IN ERROR is generated.

An attempt is made to compile some types of incorrect programs. An internal diagnostic message accompanies such an abort.

Other dayfile messages that might be produced include:

```
-SYMPL- INSUFFICIENT FL
-SYMPL- INSUFFICIENT SCM FL
-SYMPL- INSUFFICIENT LCM FL
-SYMPL- EMPTY INPUT FILE
-SYMPL- COMPILER ABORT
-SYMPL- BAD EXP CALL TO FTN
-SYMPL- BAD LOADER CALL
-SYMPL- cccccccccc COMPILED  cp secs
```

Table B-1 lists the message number and text of the compilation diagnostics. Abbreviations used in these messages are:

Abbreviation	Description
BOOL	Boolean
CHAR	Character
CHARS	Characters
CONS	Constant
DECL	Declaration
DUP	Duplicate
ERR	Error
EXPR	Expression
FUNC	Function
HEX	Hexadecimal
ID	Identifier
IFXX	Conditional compilation computation word
ILL	Illegal
PARAM	Parameter
PARENS	Parenthesis
PROC	Procedure
PROG	Program
REF	Reference
REFS	References
REPL	Replacement
SEMI	Semicolon
SPECS	Specifications
STMT	Statement
STRG	String
UNDECL	Undeclared
XDEF	External definition
XREF	External reference
/	Or

TABLE B-1. COMPILER ERROR MESSAGES

Message Number	Condition Causing Message
001	LONG ID–FIRST 12 CHARS USED
002	DUP DECL–NEW ONE OVERRIDES
003	UNDECL ID DELETED
004	ILL OCTAL/HEX CONS

TABLE B-1. COMPILER ERROR MESSAGES (cont.)

Message Number	Condition Causing Message
005	TERM MISSING
006	BAD STATUS CONS USE
007	BAD NESTING OF PARENS/ BRACKETS
008	CRUD CHAR IN INPUT
009	CHAR STRG>240 BYTES–240 USED
010	ILL ARRAY ITEM ID USE DELETED
011	ILL SWITCH ID USE DELETED
012	ILL ARRAY ID USE DELETED
013	ILL STATUS LIST ID USE DELETED
014	ILL COMMON ID USE DELETED
015	SEMI MISSING AFTER ARRAY DECL
016	CRUD AT START OF STMT DELETED
017	ILL KEYWORD USE DELETED
018	ARRAY ITEM DECL LIST LACKS END
019	DUP DECL OVERRIDES
020	ITEM DECL ID ERR
021	DECL DISCARDED–SCAN RESUMES AT SEMI
022	ITEM DECL TYPE ERR–I ASSUMED
023	ILL ITEM LENGTH–1 BYTE USED
024	SIGNED PRESET ILL FOR THIS TYPE–IGNORED
025	SCAN RESUMES AT -BEGIN-
026	MISSING SEMI
027	ITEM PRESET ERR
028	SEMI ACCEPTED AS NULL STMT
029	BASED/XDEF/XREF ARRAYS NEED ID
030	ARRAY ITEM DECL SYNTAX ERR
031	ARRAY ITEM DECL TYPE ERR
032	BAD ARRAY BOUND VALUES– ASSUMED [0:0]
033	ARRAY BOUND SYNTAX ERR
034	ARRAY ITEM DECL PARTWORD SPECS ERR–DEFAULT TAKEN
035	ARRAY ITEM DECL 1ST BIT ALIGNMENT WRONG–0 USED
036	ILL ARRAY ITEM BOUNDARY– DEFAULT TAKEN
037	TOO MANY ARRAY ENTRIES
038	TOO MANY PRESET GROUPS
039	ARRAY PRESET SYNTAX ERR
040	COMMON/XDEF/XREF–AT OUTER SCOPE ONLY
041	BAD COMMON DECL IGNORED

Message Number	Condition Causing Message
042	BAD XREF/XDEF IGNORED
043	BAD BASED DECL IGNORED
044	XDEF/XREF LIST CRUD DELETED
045	SWITCH DECL SYNTAX ERR
046	COMMON LIST SCAN RESUMES AT -ARRAY-/-ITEM-
047	STATUS DECL SYNTAX ERR
048	-END- ENDS BAD COMMON LIST
049	DEF DECL SYNTAX ERR
050	BAD FORMAL PARAM DECL
051	PROGRAM BEGINS BADLY
052	PROG DECL LACKS ID
053	PROG DECL ERR–CRUD PRECEDES SEMI
054	XDEF/XREF LIST SCAN RESUMES AT LEGAL ENTRY
055	FORMAL LABEL DECL SYNTAX ERR
056	-END- ENDS BAD XDEF/XREF LIST
057	FORMAL PROC DECL SYNTAX ERR
058	FUNC DECL LASKS ID
059	FUNC DECL TYPE ERR– I ASSUMED
060	FUNC DECL LACKS SEMI
061	SCAN RESUMES AT SEMI
062	DUP FORMAL PARAM ID IN LIST
063	DUP PARAM ID–PRIOR DECL THIS SCOPE
064	PARAM LIST SYNTAX ERR
065	PROC DECL LACKS ID
066	PROC DECL SYNTAX ERR
067	UNDECL LABEL/PROC ID
068	FORMAL ID LACKS DECL
069	PARAM NOT USED IN THIS SCOPE
070	ILL DEF ID–NO EXPANSION
071	ENTRY PROC MAY NOT CALL ITSELF
073	TOO MANY PARAM/ARRAY/ARRAY ITEM REFS
074	TOO MANY SUBSCRIPTS:SWITCH REF
075	NOT ENOUGH SUBSCRIPTS FOR ARRAY/ARRAY ITEM REFS
076	BAD SUBSCRIPT LIST
077	ILL LABEL/PROC ID USE DELETED
078	STATUS SWITCH DECL LACKS STATUS LIST ID
079	BAD LABEL USE IN STATUS SWITCH

TABLE B-1. COMPILER ERROR MESSAGES (cont.)

Message Number	Condition Causing Message
080	STATUS SWITCH–VALUE TOO LARGE
081	STATUS SWITCH–DUP CONSTANT VALUES
082	STATUS SWITCH–MISSING CONSTANT
083	BEGIN/END MISMATCH. PROBABLE DISASTER
084	IF EXPR NOT BOOL
085	WHILE EXPR NOT BOOL
086	CRUD AFTER FINAL END IGNORED
087	-DEF- ID EXPANSION NEST TOO DEEP-ID DELETED
088	YOUR -DO- HAS BEEN FOUND
089	THE -THEN- HAS BEEN FOUND
090	MISSING -DO-
091	MISSING -THEN-
092	INITIAL VALUE EXPR ERR
093	-STEP- EXPR ERR
094	-UNTIL- EXPR ERR
095	-WHILE- EXPR ERR
096	BAD -GOTO- DELETED
097	BAD REPL STMT DELETED
098	PARTWORD VALUES AFTER FIRST 3 IGNORED
099	ITEM DISCARDED–SCAN RESUMES AT COMMA
100	HANGING -IF- CLAUSE
101	HANGING -FOR- CLAUSE
102	HANGING -ELSE-
103	EXTRA END–OMITTED BEGIN FOR SUBPROGRAM ASSUMED
104	ILL UNDECL PARAM USE DELETED
105	FOR STMT: INDUCTION ID ERR
106	-IF- EXPR ERR
107	DUP XDEF/XREF DECLS FOR ID
108	XDEF PROC/FUNC: NOT FULLY DECL
109	BAD FORMAL DECL
110	REDUNDANT FORMAL DECL
111	BAD PARAM LIST
112	BOOL ILL IN ARITH CONTEXT
113	COMMON LIST LACKS -END-
114	BASED LIST LACKS -END-
115	XDEF/XREF LIST LACKS -END-
116	COMMON LIST CRUD DELETED
117	BASED LIST CRUD DELETED

Message Number	Condition Causing Message
118	BASED LIST SCAN RESUMES WITH -ARRAY-
119	-END- ENDS BAD BASED LIST
120	0 LENGTH -DEF- STRING IGNORED
121	CHAR LENGTH OMITTED–1 ASSUMED
122	BAD ARRAY ENTRY SIZE
123	BRACKET NEST TOO DEEP
124	ILL EXPR TYPE THIS LEFT SIDE
125	BAD READ FUNC
126	EXPR OP CONCATENATION ERR
127	LONG CHAR STRG–240 BYTES USED
128	BAD -LOC- FUNC
129	BAD -ABS- FUNC
130	BAD INDUCTION ID TYPE
131	NON INDUCTION ID IN -TEST-
132	-TEST- ILL OUTSIDE LOOP
133	SCAN RESUMES AT -BEGIN-/ -ITEM-/SEMI
134	READ FUNC NEEDS ID
135	DUP STATUS ID
136	SEMI ENDS COMMENT
137	CONTROL STMT SYNTAX ERR
138	CHAR NOT D/F IN REAL OR COUBLE CONSTANT
139	FORMAL PARAM PRESET ILL
140	XREF PRESET ILL
141	BLANK COMMON PRESET ILL
142	BASED ARRAY ITEM PRESET ILL
143	BAD P-FUNC
144	CHARACTER ITEM>240 BYTES - 240 USED
145	NO SUBSCRIPT FOR ARRAY ITEM - 0 USED
146	CIRCULAR DEF NAME EXPANSION - EXPANSION IGNORED
147	NO MAIN PROC FOR ENTRY PROC
148	ILLEGAL CHAR IN MACRO DEF
149	ILLEGAL IFXX COMPARE
150	TOO MANY DEF PARAMS
151	ILLEGAL CONDIT DIRECTIVE IGNORED
152	ILLEGAL VALUE PARAM-LABEL
153	ILLEGAL VALUE PARAM-ARRAY
154	ILLEGAL VALUE PARAM-PROC
155	COMMON BASED ARRAY DECL ERROR

TABLE B-1. COMPILER ERROR MESSAGES (cont.)

Message Number	Condition Causing Message
156	LABEL DECL ERROR
157	XREF SWITCH ERROR
158	UNMATCHED IFXX
159	DEF PARAM ERROR
160	([OR < NESTING TOO DEEP
161	([OR < NEST MISMATCH
162	PARAMETER TOO LONG
163	PARAMETER COUNT ERROR
164	RECOVERY AT ;
165	BAD DEF ACTUAL PARAMETER
166	BAD UNDCL PROC/LABEL LIST
167	ILL DEF PARAM USAGE
168	SORRY BUT IFXX MUST HAVE 2 PARAMS–FOR THE TIME BEING
169	ATTRIBUTE SPECIFIED TO UNKNOWN VARIABLE
170	SIMPLE ITEMS MAY NOT BE INERT/REACTIVE
171	ONLY ITEMS AND ARRAYS HAVE ATTRIBUTES
172	BAD ATTRIBUTE/LEVEL SPECIFICATION LIST
173	FAST FOR LOOP INDUCTION VARIABLE ERROR
174	BAD GLOBAL ATTRIBUTE SPEC
175	LEVEL ONLY APPLIES TO COMMON AND BASED ARRAYS
176	BAD USE OF LEVEL 3 VARIABLE
177	INDUCTION VARIABLES MUST BE SCM RESIDENT
178	WEAK ONLY APPLIES TO EXTERNAL SYMBOLS
179	ARRAY ENTRY-SIZE TOO LARGE
180	ARRAY DIMENSION TOO LARGE
181	RECURSIVE PROC/FUNC CALL NOT ALLOWED
182	ERROR IN REAL CONSTANT

PROGRAMMING SUGGESTIONS C

COMPILER

Space required for compilation is proportional to the number of symbols in the source program. Approximately five words of core are dedicated to each name in the program, in the form of a symbol table entry.

Time required for compilation is proportional to the size of the object program, in terms of the amount of syntax to be scanned. Although data declarations do not generate code, they use significant amounts of compiler time and field length, especially data presets.

Compilation time can be further reduced by judicious use of the compiler options such as suppression of object code and cross reference listings.

DEF declarations can increase readability of SYMPL source programs and facilitate changes to them. However, DEF declarations and expansions increase compilation time and field length, accordingly.

OBJECT CODE

SUBSCRIPTS

Code produced by referencing subscripted variables can be affected by the means of expressing the subscript. For example, an integer constant can be partially evaluated at compile time so that one instruction is required to access an array item (given the item is a full word); but a scalar integer variable requires four instructions to access the item. Thus, a reference to A[3] requires one instruction; but A[I], where I=3, requires four instructions to retrieve the same item.

ARRAYS

Parallel arrays are accessed more efficiently than serial arrays when an array entry exceeds one word. For arrays with one-word entries, no difference in object code speed or space is apparent. Parallel arrays, rather than serial arrays, should be used when possible. Fixed arrays are accessed more efficiently than based arrays, which require a level of indirectness to access an entry. Whenever possible, fixed arrays should be used.

COST OF ACCESSING DATA TYPES

If an array item is a full 60-bit word, access does not depend upon its type. For items which are not 60-bit words, however, type and bit position assignment affect the code required to access them, as follows:

> Signed integers are accessed more efficiently than unsigned integers. If the item is 18 bits long, the SXi instruction is used to access signed integers. Signed integer items are accessed more efficiently if they are the leftmost bits of a word. Unsigned integer items are accessed more efficiently if they occupy the rightmost bits of a word rather than the middle or leftmost bits. Boolean items are most efficiently accessed by allocating the whole word or the leftmost bit of a word rather than one bit elsewhere. Otherwise, they are accessed as unsigned integers are accessed.

FOR LOOPS

The break-even point in code generated for in-line and FOR loop code is 3-4 iterations. Of the following sequences, the second generates fewer instructions and runs faster.

```
FOR I=0 STEP 1 UNTIL 2 DO
  PWORD[I] = 0;
```

```
PWORD[0] = 0;
PWORD[1] = 0;
PWORD[2] = 0;
```

If four or more items were being set by the above sequence, the loop would have required less code but required more time.

In general, the less source code in the FOR statement, the faster it will run. Of the following code sequences, the second is faster since the loop limit is computed and the value stored only once.

```
FOR I = 0 STEP 1 UNTIL B/C DO
     PWORD[I] = K**J;

A = B/C;
D = K**J;
FOR I = 0 STEP 1 UNTIL A DO
     PWORD[I] = D;
```

One execption is that FOR loop execution time can be reduced with more source code as in the following example where the second sequence would be faster even though more code would be generated.

```
FOR I = 0 STEP 1 UNTIL 89 DO
     PWORD[I] = 0;

FOR I = 0 STEP 3 UNTIL 89 DO
     BEGIN
       PWORD[I] = 0;
       PWORD[I+1] = 0;
       PWORD[I+2] = 0;
     END
```

DATA CONVERSION

Integer-to-character conversion is byte-oriented; the character-to-integer conversion is word-oriented. When an integer item is converted to character mode, the rightmost 6-bit byte is left-justified and blank filled in the character field; yet, character-to-integer conversion is performed by right-justifying the right end of the last word of the character item and zero filling it on the left. Character field definitions can cross word boundaries. Arithmetic operations with character data, including masking, makes the code machine dependent because it reduces the string to one word.

The conversions can be circumvented by the use of bit bead functions. For example, B<0,60>FLTINGPT =INTEGER; would cause the integer to be stored in the floating point item without conversion. B<0,60> CHARACTER=INTEGER also would cause the full word to be stored in CHARACTER, not just the low-order six bits.

PROC SUBPROGRAMS

Formal parameters should be called by value whenever possible. If a procedure must reference its formal call by address parameter more than once, a local variable should be declared, set to the value of the formal parameter, and subsequently referenced instead of the formal parameter. Actual call-by-name parameters are referenced indirectly in the generated code; this level of indirectness can be overcome by evaluating the parameter once and making it local to the procedure by storing the parameter's value in a local variable.

FUNC SUBPROGRAMS

The statements under the heading PROC subprograms are true for FUNC subprograms also. When the subprogram must return a result, a function should be used rather than a procedure that returns a value. Use of the function saves two instructions. For example: a routine is needed to convert from integer to display code, with the result to be stored in one of three arrays, depending upon the section of code where the call originates. If a function is used (as in ARRAYWORD[I] = FUNCTION[INT] rather than a procedure (as in PROCED (INT); ARRAYWORD[I] = INTT), two SAi k instructions are saved per call. The saving is realized since functions return their result in register X6 rather than in a memory location.

CODING HINTS

Based array references are candidates for scratch variable storage if referenced more than once in a sequence of source code, since based array references are indirect.

When storing into many items of the same data structure (array) clustered together, those that refer to the same word of storage should be described in the same order in which they occur.

POSSIBLE OPTIMIZATIONS

The SYMPL language permits the compiler to move code to achieve optimization. SYMPL 1.2 and later versions, at the present time, do not perform global flow analysis. They do, however, perform many local optimizations including: compile-time computation of constant expressions, conversion of many multiplies to shift-and-add, and elimination of many redundant loads and stores. Therefore, if the program has any OVERLAP or REACTIVE variables, they should be declared to assure correct compilation on SYMPL 1.2 and later versions of the compiler.

In SYMPL 1.2 and later versions, if no CONTROL statements with INERT, REACTIVE, DISJOINT, or OVERLAP appear, the program is called unbehaved and is considered to adhere to SYMPL 1.1 rules, which are:

Formal parameters can destroy global variables and vice versa.

A based array can destroy all other based and fixed arrays, but a fixed array does not destroy any other arrays.

All arrays are considered reactive.

An external call can destroy all COMMON, XDEF and XREF variables.

Formal parameters can destroy each other.

There are no other interferences between variables.

These definitions are retained in SYMPL 1.2 and later versions to accommodate existing programs until correct behavior statements are inserted.

OPTIMIZATIONS POSSIBLE UNDER GLOBAL OPTIMIZATION

The compiler is permitted all the optimizations listed below.

Constant Subsumation: If a constant is assigned to a variable, replace the variable with the constant up until a point where its value may be destroyed.

Common-Expression removal: If the same expression occurs twice and none of the variables are destroyed in between, save the result of the first computation, eliminate the code for the second computation, and reference the saved value.

Removal of identities: Remove statements such as I=I; and through constant subsumation and the mechanisms of common-expression removal, the optimizer might determine that a statement is in fact an identify though this is not apparent in the source.

Code removal from loops: Recognize program flow which is a loop, whether it is a formal FOR-loop or not, and optimize any loop which is not spoiled by a branch entering from outside. Code which is invariant during the loop is moved in front of the loop.

Strength reduction: In a fastloop, certain multiply operations on the induction variable are converted to additions to a temporary variable, and certain exponentiations are similarly converted to multiplications.

This SYMPL language definition permits analysis of program flow to discover loops (including nested loops) and to determine which expressions are invalidated by forward branches. It may also analyze all procedures and functions within a module to determine which variables they use and which ones they destroy. This enables the optimizer to optimize over many function or procedure calls. Since it is possible for code to be removed over long distances in the program, the programmer must inspect the entire module to determine OVERLAP or REACTIVE behavior.

The compiler never moves code from one procedure to another. Suppose PROC Q stores B(I) and PROC P references A(J) and B(I) is based on A(J). If P calls Q, there is danger of the A(J) reference being moved past the call to Q; this is overlapped behavior and the CONTROL OVERLAP statement is required to prevent such optimization. But if the program is restructured so that P and Q are parallel (neither one calls the other), then this is not overlapped behavior. For example:

```
PROC MAIN;
    BEGIN
    ARRAY A[10]; ITEM AA(0);
    BASED ARRAY B[10]; ITEM BB(0);
      :
      PROC INIT;
      BEGIN
      AA(1) = 31;
      END #INIT#
      :
    P<B> = LOC (A);

L1:
    INIT;
    X = BB[I];
      :
    IF BOOLE THEN GOTO L1;
    END #MAIN#
```

Here the compiler might remove BB[I] from the loop, causing an error that might be difficult to locate. The statement

```
CONTROL OVERLAP A,B;
```

solves this problem. However, if the code between the INIT call and the IF BOOLE statement is converted to a procedure, the problem will not arise and no CONTROL statement is required.

Such a problem occurs frequently in programs having a separate initialization section: the program can remain well-behaved if both the initialization and the body are made into separate procedures.

Another common problem is the local based array whose pointer is manipulated by an external procedure. (The Common Memory Manager is a case in point.) Such based arrays must be declared overlapped. For example:

```
XREF PROC GETSPC;
BASED ARRAY X[100]; ITEM XX (0);
   :
GETSPC(X, 100);
Q=P<X>;
GETSPC(Y, 50);
R=P<X>;
```

Suppose the routine GETSPC is external and manages dynamic storage, and suppose that at the GETSPC(Y, 50) call, it moves block X. Now if the optimizer removes the expression P<X> and sets R to the old P<X> from the statement Q=P<X>, the result will be wrong.

The compiler can assume that GETSPC(Y) does not destroy X because X is a local, and theoretically GETSPC cannot get at X unless X is a parameter. This assumption is not of course fully correct; however, we define the language to consider this to be overlapped behavior and require the statement:

```
CONTROL OVERLAP X;
```

TREATMENT OF EXTERNALS AND COMMON

All badly-behaved and all external variables (XDEF, XREF, and COMMON) are considered destroyed by an external call. Any global flow analysis analyzes all possible flow of control resulting from an XDEF label, and considers that all variables are destroyed by entry at such a label.

```
condition word      :=   { ifeq
                           ifne
                           ifls
                           iflq
                           ifgq
                           ifgr }

control word        :=   { eject
                           list
                           nolist
                           objlst
                           pack
                           preset
                           fi
                           traceback
                           ftncall
                           fastloop
                           slowloop }

attribute           :=   { level ∧ lev list
                           inert ∧ var list
                           reactive ∧ var list
                           disjoint ∧ var list
                           overlap ∧ var list
                           weak ∧ weak list }

lev list            :=   { lev descr
                           lev list ∨ , ∨ lev descr }

lev descr           :=   { common name
                           based array name
                           ϕ }

var list            :=   { var descr
                           var list ∨ , ∨ var descr }
```

$$\underline{\text{var descr}} := \left\{\begin{array}{l}\underline{\text{array name}}\\ \underline{\text{based array name}}\\ \underline{\text{item name}}\\ \phi\end{array}\right\}$$

$$\underline{\text{weak list}} := \left\{\begin{array}{l}\underline{\text{weak descr}}\\ \underline{\text{weak list}}\ \underline{\vee}\ ,\ \underline{\vee}\ \ \underline{\text{weak descr}}\end{array}\right\}$$

$$\underline{\text{weak descr}} := \left\{\begin{array}{l}\underline{\text{array name}}\\ \underline{\text{based array name}}\\ \underline{\text{function name}}\\ \underline{\text{item name}}\\ \underline{\text{label name}}\\ \underline{\text{proc name}}\\ \underline{\text{switch name}}\end{array}\right\}$$

ifeq	:=	mark	⌋	IFEQ	⌊	mark
ifne	:=	mark	⌋	IFNE	⌊	mark
ifls	:=	mark	⌋	IFLS	⌊	mark
iflq	:=	mark	⌋	IFLQ	⌊	mark
ifgq	:=	mark	⌋	IFGQ	⌊	mark
ifgr	:=	mark	⌋	IFGR	⌊	mark
eject	:=	mark	⌋	EJECT	⌊	mark
list	:=	mark	⌋	LIST	⌊	mark
nolist	:=	mark	⌋	NOLIST	⌊	mark
objlst	:=	mark	⌋	OBJLST	⌊	mark
pack	:=	mark	⌋	PACK	⌊	mark
preset	:=	mark	⌋	PRESET	⌊	mark
fi	:= {	mark	⌋	FI	⌊	mark }
		mark	⌋	ENDIF	⌊	mark
traceback	:=	mark	⌋	TRACEBACK	⌊	mark
ftncall	:=	mark	⌋	FTNCALL	⌊	mark
fastloop	:=	mark	⌋	FASTLOOP	⌊	mark
slowloop	:=	mark	⌋	SLOWLOOP	⌊	mark
level	:=	mark	⌋	LEVEL $\left\{\begin{array}{c}1\\2\\3\end{array}\right\}$	⌊	mark
inert	:=	mark	⌋	INERT	⌊	mark

```
reactive      := ⌋ mark     REACTIVE  ⌊ mark

disjoint      := ⌋ mark     DISJOINT  ⌊ mark

overlap       := ⌋ mark     OVERLAP   ⌊ mark

weak          := ⌋ mark     WEAK      ⌊ mark

                    ⎧ array dec          ⎫
                    ⎪ based dec          ⎪
                    ⎪ common dec         ⎪
                    ⎪ def dec            ⎪
                    ⎪ entry dec          ⎪
                    ⎪ func dec           ⎪
                    ⎪ item dec           ⎪
                    ⎪ label dec          ⎪
                    ⎪ proc dec           ⎪
declaration   :=    ⎨ status dec         ⎬
                    ⎪ switch dec         ⎪
                    ⎪ xdef dec           ⎪
                    ⎪ xref dec           ⎪
                    ⎪ formal array dec   ⎪
                    ⎪ formal based dec   ⎪
                    ⎪ formal func dec    ⎪
                    ⎪ formal item dec    ⎪
                    ⎪ formal label dec   ⎪
                    ⎩ formal proc dec    ⎭

                    ⎧ compound statement      ⎫
                    ⎪ exchange statement      ⎪
                    ⎪ for statement           ⎪
                    ⎪ goto statement          ⎪
                    ⎪ if statement            ⎪
statement     :=    ⎨ labeled statement       ⎬
                    ⎪ proc call statement     ⎪
                    ⎪ replacement statement   ⎪
                    ⎪ return statement        ⎪
                    ⎪ stop statement          ⎪
                    ⎩ test statement          ⎭
```

GLOSSARY F

ARITHMETIC EXPRESSION – An expression that yields a numeric value.

BASED ARRAY – A structure that can be superimposed over any area of memory during program execution. No storage is allocated for a based array during compilation; rather the compiler creates a specific pointer variable compiled with an undefined value. Based arrays are used when the position of an array is not known at load time.

BEAD FUNCTION – A function that accesses consecutive bits or characters of an item.

BOOLEAN EXPRESSION – An expression that yields a Boolean value of TRUE or FALSE.

DELIMITER – A character that is used to separate and organize data items or statements. SYMPL characters classified as marks serve as delimiting characters.

ENTRY POINT – A location within a procedure or function that can be referenced from a calling program. Each entry point has a name with which it is associated.

EXCHANGE STATEMENT – A statement that causes the exchange of values of the left-hand and right-hand sides of the statement.

EXPRESSION – A sequence of identifiers, constants, or function calls separated by operators and parentheses. The evaluation of an expression yields a value.

EXTERNAL REFERENCE – A reference in one module to an entry point in another module. Throughout the loading process, the loader matches externals to the correct entry points. External references are specified by the XREF statement.

EXTERNAL SUBPROGRAM – A subprogram that is compiled as a separate module.

FASTLOOP – A type of FOR statement where the test and branch is at the end of the loop. Fastloops always execute at least once. Contrast with slowloop.

FUNCTION – A subprogram used within an expression. It returns a value through its name. The text of a function must contain an assignment statement that assigns a value to the function name. A function can also return values through its parameters. Contrast with procedure and main program.

IDENTIFIER – A string of 1 through 12 letters, digits, or $ beginning with a letter ($ is considered to be a letter). This manual uses the term identifier to indicate a programmer-defined entity. Contrast with reserved words.

INDUCTION VARIABLE – A scalar that is used as the counter for the loop in a FOR statement.

LOGICAL OPERATOR – An operator that works with Boolean values and yields a Boolean result. Contrast with masking operator, numeric operator, and relational operator.

MAIN PROGRAM – A module that consists of a main program header followed by a series of declarations and one statement (usually compound) and ended by a TERM statement. Contrast with function, procedure, and subprogram.

MASKING OPERATOR – An operator that performs bit-by-bit operations that yield numeric results. Contrast with logical operator, numeric operator, and relational operator.

MODULE – A separately compiled main program or subprogram. Compilation of a module is terminated whenever a TERM statement is encountered.

NUMERIC OPERATOR – An operator that performs arithmetic operations to yield numeric results. Contrast with logical operator, masking operator, and relational operator.

PARALLEL ALLOCATION – The first words of each array entry are allocated contiguously, followed by the second words of each entry, and so forth. Contrast with serial allocation.

P-FUNCTION – A function that references the pointer variable for a based array.

POINTER VARIABLE – The variable created by the compiler for a based array. The pointer variable is set by the P-function.

PROCEDURE – A subprogram that can, but need not, return values through any of its parameters. It is called when its name or one of its alternative entry points is referenced. Contrast with function and main program.

RELATIONAL OPERATOR – An operator that works with arithmetic or character operands to produce a Boolean result. Contrast with logical operator, masking operator, and numeric operator.

REPLACEMENT STATEMENT – A statement that assigns a value to a scalar, subscripted array item, P-function, bead function, or function name.

RESERVED WORDS – Identifiers that have predefined meaning to the SYMPL compiler.

SCALAR – An item that is not in an array. An ITEM declaration outside an array defines a scalar.

SCOPE OR VARIABLE – The set of statements in which the declaration of the variable is valid.

SERIAL ALLOCATION – All the words of one array entry are allocated contiguously. Contrast with parallel allocation.

SLOWLOOP – A type of statement where the test and branch is at the beginning of the loop. Slowloops need not execute at all. Contrast with fastloop.

SUBPROGRAM – A function or procedure. Subprograms can be compiled as separate modules. Contrast with main program.

TYPE – The representation of data. Data can be type integer, unsigned integer, real, character, Boolean, or status.

WEAK EXTERNAL – An external reference that is ignored by the loader during library searching and cannot cause any other program to be loaded. A weak external is linked, however, if the corresponding entry point is loaded for any other reason.

XDEF DECLARATION – A declaration that generates an entry point that can be used by the loader. It is used in the declaring program to define an identifier as external. Storage is allocated for the identifier. Contrast with XREF declaration.

XREF DECLARATION – A declaration that generates an external reference to the specified identifier. It is used in the referencing program. Use of XREF implies that the identifier has been declared to be external in another program. No storage is allocated for the identifier. Contrast with XDEF declaration.

INDEX

CUT ON THIS LINE

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: SYMPL Version 1 Reference Manual

PUBLICATION NO. 60496400 REVISION D

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

General comments:

FROM NAME: ______________________ POSITION: ______________________

COMPANY NAME: ______________________

ADDRESS: ______________________

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

STAPLE

FOLD FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Publications and Graphics Division
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD FOLD

STAPLE STAPLE