96836100

CONTROL DATA CORPORATION

CDC® CYBER CROSS SYSTEM VERSION 1 PASCAL COMPILER REFERENCE MANUAL

CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Manual released.
(12/75)	
B	Manual revised to reflect NOS/BE 1.1 changes (comprised primarily of PASCAL enhancements) and
(4/76)	manual name change from CCP Support Software to CYBER Cross System.
C	Manual revised to incorporate CYBER Cross NOS R6.
(8/79)	
D	Revised at PSR level 528 to reflect a change in symbol table size and for various technical corrections.
(10/31/80)	
Publication No. 96836100	

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
Front Cover	--
Title Page	--
ii	D
iii/iv	D
v/vi	D
vii	B
viii	B
1-1 thru 1-3	B
2-1 thru 2-13	B
2-14	D
2-15 thru 2-20	B
2-21	D
2-22 thru 2-27	B
3-1 thru 3-8	B
4-1	B
4-2	D
4-3 thru 4-5	B
5-1 thru 5-4	B
6-1 thru 6-3	B
7-1 thru 7-6	B
7-7	D
7-8	B
Glossary-1	B
A-1 thru A-4	B
B-1	B
B-2	B
C-1 thru C-4	B
D-1	B
E-1	D
F-1 thru F-3	B
G-1	D
H-1	B
H-2	B
Index-1 thru Index-3	B
Comment Sheet	D
Mailer	-
Back Cover	-

PREFACE

This manual describes the PASCAL programming language for the CONTROL DATA® CYBER 18 series computer and the CDC 255x Host Communications Processors. PASCAL operates on the CYBER 170/70/6000 computers and generates object code suitable for execution on a CYBER 18 series computer or a CDC 255x processor. The PASCAL compiler is a component of the CYBER Cross System operating under control of the NOS or NOS/BE operating system.

This manual is provided to serve both as an introduction and a reference to the PASCAL programming language. The descriptions are presented utilizing a syntactical notation coupled with a semantic discussion of the various language elements. Appendix A provides a brief summary of syntax descriptions for the PASCAL language elements that appear in this manual.

Detailed information can be found in the listed publications. The publications are listed alphabetically within groupings that indicate relative importance to readers of this manual.

The NOS and NOS/BE manual abstracts are instant-sized manuals containing brief descriptions of the contents and intended audience of all NOS and NOS product set manuals, and NOS/BE and NOS/BE product set manuals, respectively. The Software Publications Release History can be useful in determining which revision level of software documentation corresponds to the Programming Systems Report (PSR) level of installed site software.

The following publications are of primary interest:

Publication	Publication Number
CYBER Cross System Build Utilities Version 1 Reference Manual	60471200
CYBER Cross System Version 1 Reference Manual	96836000
Mass Storage (MS) FORTRAN Version A/B Reference Manual	60362000
Mass Storage Operating System (MSOS) Reference Manual	96769400
NOS Version 1 Reference Manual, Volume 1 of 2	60435400
NOS/BE 1 Reference Manual	60493800

The following publications are of secondary interest:

Publication	Publication Number
NOS Version 1 Installation Handbook	60435700
NOS/BE Version 1 Installation Handbook	60494300
NOS Version 1 Manual Abstracts	84000420
NOS/BE Version 1 Manual Abstracts	84000470
Software Publications Release History	60481000

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.

CONTENTS

TABLES

The SYMPL language is similar to the JOVIAL language, which was derived from the ALGOL-58 language. Consequently, it has many similarities with ALGOL and ALGOL-like programming languages such as PL/I, although it has features not found in these better known languages. SYMPL also has similarities with the COMPASS assembly language and the FORTRAN Extended compiler language. SYMPL statements provide Boolean and algebraic capabilities; the declarations provide the data structures of other languages.

Since SYMPL is a systems programming language, it does not include input/output facilities. When a SYMPL subprogram is called from a FORTRAN Extended main program, however, the FORTRAN language PRINT statement capabilities can be used within the subprogram for debugging purposes.

Of more significance for input and output, however, is the fact that the calling sequence conventions for FORTRAN Extended and SYMPL are alike. Both COMPASS and FORTRAN Extended routines interface easily with SYMPL.

CHARACTERISTICS OF SYMPL

The SYMPL language is characterized by:

Reserved words

Orientation toward manipulation of bits and 6-bit bytes as well as words

Free-form program format, although good programming practices and coding conventions advise use of a more rigid structure

Nonexecutable declaratives that describe data structure and use

Nonexecutable compiler-directing statements

Executable statements that describe procedures to be carried out

Block structure in which the reserved words BEGIN and END delimit compound statements and declarations

Loader capabilities for interprogram communication available through SYMPL include: the COMMON declaration that produces named or blank common blocks; the XREF declaration that produces loader external references; and the XDEF declaration that produces loader entry points.

SYMPL COMPARED WITH FORTRAN

This user guide illustrates many SYMPL concepts by comparing SYMPL with FORTRAN Extended. Figures 1-1 and 1-2 show two jobs that produce the same results through FORTRAN Extended and SYMPL. Figure 1-1 shows a job deck containing a FORTRAN Extended program that generates and prints the first 10 Fibonacci numbers. (A Fibonacci number is defined as the sum of the two immediately preceding Fibonacci numbers.)

a. Job deck

```
job statement
FTN,R=0.
LGO.
7/8/9
  PROGRAM FIBON(OUTPUT)
  INTEGER L(10)
  DATA L(1),L(2) /1,1/
  LIMIT=10
  PRINT 4,LIMIT
4 FORMAT (*1FIRST *,I2,
         * FIBONACCI NUMBERS*)
  DO 1 N=3,LIMIT
  L(N)=L (N-1) + L(N-2)
1 CONTINUE
  DO 2 N=1,LIMIT
  PRINT 3,L(N)
2 CONTINUE
3 FORMAT(1H ,I10)
  STOP
  END
6/7/8/9
```

b. Output from program FIBON

```
FIRST 10 FIBONACCI NUMBERS
         1
         1
         2
         3
         5
         8
        13
        21
        34
        55
```

Figure 1-1. FORTRAN Extended Fibonacci Numbers Example

Figure 1-2 illustrates the same task as figure 1-1. Since the SYMPL subprogram uses FORTRAN 'output, a FORTRAN main program is required to control the environment in which the SYMPL subprogram executes. The XREF statement is necessary to allow access to library procedures which perform output. The SYMPL subprogram statements are arranged in the order of the FORTRAN program statements. Current Control Data coding standards require the subprogram to be structured as shown in figure 1-3 and to include comments.

Syntax differences between SYMPL and FORTRAN include the following SYMPL conventions:

All statements must be terminated by a semicolon.

BEGIN and END delimit a compound statement that can contain other elementary or compound statements.

```
job statement
FTN,R=0.
SYMPL.
LGO.
7/8/9
  PROGRAM MAIN(OUTPUT)
  CALL FIBON
  STOP
  END
7/8/9
  PROC FIBON;
    BEGIN
    XREF BEGIN PROC PRINT; PROC LIST;
                          PROC ENDL; END
    DEF LIMIT #10#;
    ITEM N I;
    ARRAY [1:LIMIT]; ITEM L=[1,1];
    PRINT("(*1FIRST *,I2,* FIBONACCI
                          NUMBERS* ,//)");
        LIST(LIMIT);
        ENDL;
    FOR N=3 STEP 1 UNTIL LIMIT DO
        L[N]=L[N-1] + L[N-2];
    FOR N=1 STEP 1 UNTIL LIMIT DO
        BEGIN
        PRINT("(1H ,I10)");
            LIST(L[N]);
            ENDL;
        END
    END
  TERM
6/7/8/9
```

Figure 1-2. SYMPL Fibonacci Numbers Example

Reserved words exist; in figure 1-2, the following reserved words are used: PROC, BEGIN, XREF, END, DEF, ITEM, ARRAY, FOR, STEP, UNTIL, DO, and TERM.

A subprogram is called by the program name itself, without a preceding CALL.

Spaces are significant and can be replaced by, or accompanied by, one or more blanks or comments.

Comments are delimited by the mark # when the ASCII character set is used or ≡ when a CDC character set is used.

Brackets delimit array subscripts.

```
PROC FIBON;
BEGIN
XREF
      BEGIN
      PROC PRINT;
      PROC LIST;
      PROC ENDL;
      END
DEF LIMIT #10#;
ITEM N I;
ARRAY [1:LIMIT];
      ITEM L = [1,1];

PRINT("(*1FIRST *,I2,* FIBONACCI NUMBERS*)");
LIST(LIMIT);
ENDL;

FOR N=3 STEP 1 UNTIL LIMIT DO
      L[N]=L[N-1] + L[N-2];

FOR N=1 STEP 1 UNTIL LIMIT DO
      BEGIN
      PRINT("(1H ,I10)");
      LIST(L[N]);
      ENDL;
      END
END
TERM
```

Figure 1-3. PROC FIBON in Recommended Program Format

Language differences between SYMPL and FORTRAN include the following SYMPL conventions:

All variables must be declared, even those used only for loop control.

External subroutines are required for output.

FOR loops can have negative step increments.

IF statements have the form IF. . . THEN. . . ELSE.

Symbolic constants are allowed.

Array items are referenced by item name, not array name.

The SYMPL language consists of reserved words, programmer-supplied words, and expressions. These, in turn, are composed of characters from the SYMPL character set. The remainder of this section discusses each of these basic language elements.

SYMPL CHARACTER SET

The SYMPL character set is limited to 55 characters:

Letters A through Z and $ ($ is considered to be a letter)

Digits 0 through 9

Marks + - * / = [ ] ( ) < > " # . , : ; and blank

Other characters in the computer character set (appendix A) can appear in a SYMPL program only within a character constant or a comment.

Input and output of the SYMPL marks is complicated by the different character sets available on keypunches, terminals, and printers. Not all 026 keypunches have the same characters written on the top of keys; not all characters appear on key caps of either terminals or keypunches. A character that is keypunched for a constant as ' might appear on printed output as " or ≠, depending on the type of printer.

The two marks most frequently confused among the SYMPL character set are those used to delimit comments and some types of constants. For these functions, SYMPL requires the display code values of 60 and 64, respectively. Appendix A of most CYBER 170 software manuals shows the CDC standard character set that can be used to determine which keys must be used to obtain the punch combination for the required display code.

Delimiter For	Display Code Value	ASCII Graphic	CDC Graphic
Comment	60	#	≡
Character Constant	64	"	≠ or '

In this manual, an ASCII input device and an ASCII printer are assumed.

The marks + - * and / have the same meaning in arithmetic expressions as they do in other languages, with ** representing exponentiation and = = representing interchange. The marks , and . have customary meanings. The mark = represents replacement, as in FORTRAN Extended (not as in PL/I).

The marks [ ] ( ) and < > are explained below:

[ ] Balanced brackets delimit a subscript of an array.

<> Balanced angle brackets delimit arguments for the based array P function and the bead functions B and C.

( ) Balanced parentheses delimit arguments of a function, procedure, or DEF statement. They also group expressions and denote a call-by-value argument. As in other languages, parentheses can be used to improve readability of expressions or to force a specific evaluation order within expressions.

The two marks # and " must be used in pairs.

Paired pound signs delimit:

Comment

Character string of a DEF statement

" Paired quote marks delimit a character or status constant.

The remaining marks are used as follows:

; Semicolon terminates each declaration and executable statement, and most compiler-directing statements.

: Colon is used to:

Separate bounds of array dimension

Terminate a label

Define a status constant

RESERVED WORDS

SYMPL is a reserved-word language. A complete list of the 50-plus reserved words appears in the SYMPL Reference Manual. In this user guide, reserved words are introduced as the appropriate language element is described. Reserved words identify elements of the language. Examples are PROC and PRGM which signify program headers, ITEM and ARRAY which describe data items, and IF and ELSE which form part of executable statement syntax.

Words appearing in capital letters in statement formats presented in this manual are, for the most part, reserved words. A few words or letters required in some circumstances are not reserved words. Specifically, the following are not reserved, although good programming practice restricts use of these words to situations in which meaning cannot be confused:

Data descriptions: B, I, U, S, R, C.

Control words of the CONTROL compiler-directing statement, such as EJECT, NOLIST, PRESET, PACK, and IFEQ.

PROGRAMMER-SUPPLIED IDENTIFIERS

Identifiers are programmer-supplied names that are analogous to COMPASS names and FORTRAN variable names. Identifiers cannot be constructed through micro substitution or concatenation, however, as they can in COMPASS.

Identifiers must have these characteristics:

First character must be a letter or $

Contain 1 through 12 letters, digits, or $

Must not duplicate a reserved word

Although SYMPL identifiers can have 12 characters, it is good programming practice to limit identifier length to 10 characters. This restriction allows efficiencies in tables constructed by the compiler.

Examples of valid identifiers are:

I

X1

$IGN

SEMAPHORE

FIRSTBIT

Examples of invalid identifiers are shown in table 2-1.

TABLE 2-1. INVALID IDENTIFIERS

Identifier	Why Invalid
LIM	Reserved word
1AJ	Does not begin with letter or $
LAB"I"	Contains marks
↑↑00017	↑↑ are not SYMPL characters
FIRST CASE	Contains invalid blank
TEST	Reserved word
OPEN.RM	Contains mark

EXPRESSIONS

Expressions are used within statements. SYMPL expressions are similar to those of other languages in that they are sequences of identifiers, constants, or function calls separated by operators and parentheses. Two types of expressions are:

Arithmetic expressions that yield numeric values.

Boolean expressions that yield Boolean values of TRUE or FALSE.

ARITHMETIC EXPRESSIONS

Arithmetic expressions are used in replacement statements such as the following in which identifier A receives the value of the evaluated expression:

A=arithmetic expression;

An operand in an arithmetic expression can be any of the following:

Constant.

Variable defined as data type I, U, S, R, or C. Variables can be scalars (full 60-bit word for each item) or fields in an array (number of bits determined by array declaration) or parts of a scalar or field indicated by a bead function (a bead function extracts bits or characters from an array item or scalar).

Function call.

Boolean data cannot be used in arithmetic expressions.

All manipulation of variables takes place in full words, with SYMPL aligning a partial word field in a full word before performing the expression evaluation. Alignment is as shown in table 2-2. When data of different types is used in a simple expression, the system performs conversions as necessary. The SYMPL Reference Manual contains full details of conversion.

TABLE 2-2. DATA ALIGNMENT

Data Type	Alignment
C	Left-justified and adjusted to one word length. Data less than 10 characters is blank filled; data longer than 10 characters is truncated to 10 characters.
I	Right-justified with sign extension.
U, S	Right-justified.
R	Real data always occupies a full word and need not be realigned.

When character data is used in arithmetic expressions, only a single word of characters is involved. Any character data used as an integer is assumed to be an integer; the leftmost bit is the sign bit, and other bits are the integer value. No realignment takes place when character data becomes integer data, unless there is less than 10 characters; character data less than 10 characters is shifted right to normal integer position and zero filled. When an integer is converted to character data, however, the rightmost 6 bits of the integer are assumed to be a single character; and they are left-justified and blank filled. Any other bits in the integer are ignored. With this exception, conversions are standard for mixed data types.

The operators in an arithmetic expression can be arithmetic or logical. For the most part, character data is used only with logical operators.

Arithmetic Operators

The arithmetic operators are:

Unary operators + - and the intrinsic function ABS(exp)

Binary operators + - * / and **

A series of operators are evaluated according to FORTRAN precedence rules in which evaluation proceeds in the following order; parentheses can force a different order:

** (exponentiation)

* or / (multiplication or division)

+ or - (addition or subtraction)

When an integer is divided by another integer, the quotient is truncated without rounding. For example, the following statements produce WORD=2 and BIT=48:

```
ITEM BIT, WORD, I;
I=18;
WORD=I/10+1;
BIT=6 * (I-I/10 * 10);
```

Exponentiation is always performed in-line for all powers of two. Other small integer powers might be performed in-line, depending on compiler optimization. Integer multiplication and division by a power of two are performed in-line and are accurate to 60 bits signed.

Masking Operators

The masking operators of arithmetic expressions perform bit-by-bit operations that yield numeric values. The operators, in order of precedence, are:

LNO Complement (set 0 to 1, or set 1 to 0)

LAN Logical product (set to 1 if both bits 1)

LOR Inclusive OR (set to 1 if either or both bits is 1)

LXR Exclusive OR (set to 1 if bits are unlike)

LIM Imply (set to 1 if first operand is 0, or if first and second operands are both 1)

LQV Equivalence (set to 1 if both bits alike)

These operands work with the full word containing a scalar. More powerful masking operations result when the items are part-word array items or bead functions as described in section 5.

An example of logical product use of a 12-bit mask of zeros that sets the twelve low order bits to zero in ABCDEFGHIJ is:

```
ITEM N C(10)="ABCDEFGHIJ";
ITEM MASK2=-O"7777";
N=N LAN MASK2;
```

The following example shows exclusive OR use that sets X=0 only if A=B:

```
ITEM A,B,X;
X=A LXR B;
```

Character data used with masking operators always involves 60 bits. Shorter strings are left-justified and blank-filled; longer strings are truncated to ten characters.

BOOLEAN EXPRESSIONS

Boolean expressions are rules for determining logical values. Such expressions always yield Boolean results; that is, the result is always TRUE or FALSE. Boolean expressions are used primarily in statements that test a condition, such as:

```
IF Boolean-expression THEN . . .

FOR I=0 STEP 1 WHILE Boolean-expression DO . . .
```

A Boolean expression also can be used as the right-hand side of a replacement statement if the type of the left-hand side identifier is Boolean, as in:

```
ITEM ERRORS B, CHARCOUNT;
ERRORS=CHARCOUNT GR 7;
```

Another use of Boolean expressions is to manipulate absolute values. In the following example, SIGNE is TRUE if NUMB is less than 0. After the absolute value is obtained, if SIGNE is TRUE, NUMB is reset to negative:

```
ITEM NUMB, SIGNE B;
SIGNE=NUMB LS 0;
NUMB=ABS (NUMB);
.
.
.
IF SIGNE THEN NUMB= -NUMB;
```

Two types of operators that can be used in Boolean expressions classify the expressions:

Logical operators AND, OR, and NOT classify the expression as a logical Boolean expression.

Relational operators EQ, GR, LS, GQ, LQ, and NQ classify the expression as a relational Boolean expression.

In the following example of a relational Boolean expression, OK is TRUE if A is greater than Q:

```
ITEM A, Q, OK B;
OK=A GR Q;
```

In the following example, relational Boolean expressions and logical Boolean expressions are combined. OK is TRUE if A is greater than Q and A is not 0:

```
ITEM A, Q, OK B;
OK=(A GR Q) AND NOT (A EQ 0);
```

Logical Operators

The logical operators for Boolean expressions are identical to the FORTRAN logical operators, although they are written without the decimal point delimiters. They are implemented in SYMPL by tests such as the ZR instruction of COMPASS.

In contrast to the masking operators of arithmetic expressions, the logical operators of Boolean expressions work with one Boolean value (TRUE or FALSE) versus another, as in:

```
IDENT1 OR IDENT2
```

The Boolean logical operators, in order of highest to lowest precedence, are:

NOT	Logical negation (TRUE if neither TRUE)
AND	Logical conjunction (TRUE if both TRUE)
OR	Logical disjunction (TRUE if either TRUE)

During execution, evaluation of a Boolean expression proceeds only as long as needed to determine the result; evaluation terminates when partial evaluation satisfies the expression. For example:

```
B = (I EQ I/6*6) OR (NAME EQ "ABC");
```

If I is a multiple of 6, B is TRUE without further evaluation.

Relational Operators

The relational operators specify a comparison between two arithmetic expressions or character operands. The relational operators for Boolean expressions are equivalent to FORTRAN relational operators, although the mnemonics of the operators differ. These operators are used only with arithmetic expressions, as in:

```
IDENT3 NQ 17
```

The relational operators are:

EQ	Equals
GR	Greater than
LS	Less than
GQ	Greater than or equal to
LQ	Less than or equal to
NQ	Not equal

During execution, character values in the arithmetic expression of a relational Boolean expression are compared in-line if neither value crosses a word boundary. If either crosses a word boundary, a call to a SYMPL library routine is compiled with attendant increase in instruction execution time.

Character strings of unequal length can be compared; SYMPL expands the shorter with blank padding to the length of the longer before comparing. For example, at the end of the sequence shown in figure 2-1, AB, BC, and AC have the value TRUE.

```
ITEM A C(2)="XX",
      B C(4)="XX  ",
      C C(6)="XX    ";
ITEM AB B, BC B, AC B;
AB=A EQ B;
AC=A EQ C;
BC=B EQ C;
```

Figure 2-1. Unequal Length Character String Example

STATEMENTS

Statements in a SYMPL program can be classified by syntax or use.

Statement use is described by the terms declaration and executable statement.

A declaration defines data or subprograms and also directs the compiler.

An executable statement specifies the operations to be carried out.

Statement syntax is described by the terms elementary and compound.

An elementary statement consists of a single language statement terminated by a semicolon.

A compound statement begins with the reserved word BEGIN; it contains zero, one, or more elementary or compound statements, and it ends with the reserved word END. One compound statement is considered to be a single statement.

Classification of a statement as compound does not affect its use; that is, a compound statement can be part or all of either a declaration or an executable statement.

Elementary statements begin with a reserved word or a programmer-supplied identifier. Examples of elementary statements are shown below. Reserved words in these examples are: PROC, GOTO, CONTROL, ITEM, IF, LS, THEN, DO, and LAN.

```
PROC FIRSTONE (A, B, C);

GOTO LABELABC;

CONTROL NOLIST;

ITEM SIZEREC I=350;

IF A LS B THEN C=D;

DO XX [I]=9-I;

P=R LAN T;

MYPROCALL;
```

Compound statements form a single unit. They can be used in most places where an elementary statement can be used. One of the most common occurrences of a compound statement is in the declaration of a procedure. (Procedures are similar to FORTRAN subroutines.) The syntax of a procedure states that a procedure is declared by a procedure header followed by optional declarations followed by a single statement. Since the single statement can be a compound statement, a procedure has virtually unlimited length. For example:

```
PROC LONGONE;                 Procedure header
BEGIN
    ITEM I, J;
    XREF ARRAY K;             Single compound
        .                     statement
        .
        .
END
```

The compound statement structure can be part of a declaration, as in:

```
XDEF
BEGIN
    ITEM A;
    ITEM B;
    ITEM C;
END
```

The same three items could be declared as externals with three elementary declarations, as in:

```
XDEF ITEM A;
XDEF ITEM B;
XDEF ITEM C;
```

In many instances a compound statement must be written to perform several operations as a single logical unit. The syntax of a FOR statement, for example, states that a single statement must follow the reserved word DO. To perform three arithmetic replacement operations with a single FOR statement, the single statement following DO must be compound, as in:

```
FOR I=4 STEP 1 UNTIL 10 DO
    BEGIN
    A=B;
    C=D;        Single compound statement
    E=F;
    END
```

Single elementary FOR statement

Another instance of compound statements deals with arrays. Array declaration syntax states that the one ITEM declaration immediately following the ARRAY declaration is a named item in that array. When more than one named item occurs within the array, the ITEM declaration can be a compound statement, as in:

```
ARRAY A[0:2];
    BEGIN
    ITEM AA;
    ITEM AB;
    ITEM AC;
    END
```

The same ARRAY declaration can be written using the abbreviated format for an ITEM declaration, as in:

```
ARRAY A[0:2];
    ITEM AA, AB, AC;
```

Notice that individual declarations or executable statements within a compound statement are terminated by a semicolon, including those immediately preceding END. Statements within a compound statement are written the same way as though they were outside the compound statement context. The words BEGIN and END are reserved words and are not terminated by semicolons.

DECLARATIONS

Declarations are required in a SYMPL program to define the type and use of data and to define other entities used in the program. Each declaration begins with a reserved word. Table 2-3 shows reserved words which begin declarations.

TABLE 2-3. RESERVED WORDS THAT BEGIN DECLARATIONS

Word	Use
ITEM	Defines an item, its characteristics and, optionally, its value.
ARRAY	Defines an array, its structure, and optionally, its values for direct reference.
BASED ARRAY	For indirect reference, defines an array, its structure, and optionally the value of each item in the array.
LABEL	States that a label name is used locally as a label in the case of a duplicate name outside the subprogram when the label name has not yet been declared.
STATUS	Defines names to be associated with compiler-assigned integer values.
SWITCH	Defines a list of label names to be associated with compiler-assigned integer values.
COMMON	Defines a storage block for reference by external subprograms.
PROC	Begins a procedure subprogram to be executed when the procedure is called.
FUNC	Begins a function subprogram that results in associating a single value with the function name when the function is called during execution.
ENTRY	Defines an alternate entry point for a subprogram.
XDEF	States that a subprogram, data or switch is to be accessible external to this module.
XREF	Identifies declarations defined in an externally compiled subprogram.
DEF	Defines character strings or variables to be substituted during compilation.
CONTROL	Declares actions the compiler is to take at the time the statement is executed.

Declarations and executable statements can be intermixed in a program. However, a specific requirement concerns the placement of some declarations. For example, an item must be declared before it is referenced, and a function must be declared before it is called. A procedure, on the other hand, can be called before it is declared. These differences and requirements of each declaration are explained where each declaration is discussed in depth.

The following examples show the use of various declarations:

Definition of an item with a preset value:

```
ITEM PI R=3.14159;
```

Definition of an array and its structure:

```
BASED ARRAY A [0:4,3:5] P(2);
    BEGIN
    ITEM AA C(0,0,7)=["POS=",,,,"MAX="];
    ITEM BB I(0,42,18)=[5(4)];
        .
        .
        .
    END
```

Assignment of special properties:

```
STATUS MONTH JAN, FEB, MAR, APR;
```

Definition of a storage block that can be referenced externally:

```
COMMON INFO;
```

Specification of a subprogram:

```
FUNC ROUND(INNUM);
ITEM INNUM;
ROUND=(INNUM+9)/10;
```

Identification of local labels:

```
LABEL CASE3,CASE4;
```

Character string substitution during compilation:

```
DEF OFF #0#;
```

Conditional assembly:

```
CONTROL IFEQ OPSYS,"NOS";
```

EXECUTABLE STATEMENTS

Executable statements specify the operations to be carried out within the program using the elements defined in declarations. These statements execute in the order they appear in the program, allowing for transfer of control as a result of an executing statement. A complete list of executable statements is shown in table 2-4.

STATEMENT LABELS

A label is an identifier used to name a statement. Any executable statement in a SYMPL program can be labeled. Labels on declarations refer to the following executable code.

Labels are referenced by GOTO statements, which transfer control to the named label. SYMPL has neither an assigned GO TO statement nor a CASE statement such as are available in other languages. A feature similar to the computed GO TO statement of FORTRAN is provided in SYMPL by switches.

The format of a label is:

name:

name Identifier of 1 through 12 letters, digits, or $ that does not duplicate a reserved word or another identifier in the subprogram. The colon must immediately follow the last character.

TABLE 2-4. EXECUTABLE STATEMENTS

Statement	Use
Assignment statements such as A=B+C;	Replace item to left of = with value obtained by evaluating the expression to the right of =.
Exchange statements such as D==E;	Interchange the values of D and E.
Procedure call statements such as MYPROCCALL;	Cause execution of procedure named.
GOTO statement	Transfers control to the labeled statement specified.
FOR. . .STEP. . .DO. . . WHILE/UNTIL. . . statement and its associated TEST statement	Cause repetitive execution during specified conditions.
IF. . .THEN. . .ELSE. . . statement	Conditional execution depending on circumstances specified.
RETURN statement	Ends a function subprogram or procedure subprogram.
STOP statement	Terminates program.

The example in figure 2-2 illustrates the use of a label named FINAL.

```
PROC TESTADD;
BEGIN
    ITEM A, B, C, D;
    IF A GR B
    THEN
        GOTO FINAL;
    ELSE
        C=B-A;
    FINAL: D=C;
END
```

Figure 2-2. Label Example

A LABEL declaration can be used to declare a label. In some instances, it is required. When the compiler encounters a statement that references a label, it links the reference to the last declared label name whether or not that label was in the same procedure. If the label name has not yet been used within the procedure and a duplicate label name exists outside the procedure, a LABEL declaration is required to transfer control to the correct labeled statement within the procedure.

A LABEL declaration has the following format:

LABEL name,name, . . . ;

name Label that is to be declared subsequently.

In the example in figure 2-2, if FINAL were a label name outside this procedure it would be necessary to include a LABEL declaration within PROC TESTADD:

```
PROC TESTADD;
    BEGIN
    LABEL FINAL;
        .
        .
        .
```

Since any program statement can be labeled, a label statement can be labeled. This practice is recommended for program clarity when labels have different purposes. In the example in figure 2-3, the label LL is an exit from previous code, and label MM is the beginning of a loop.

```
        .
        .
        .
    GOTO LL;
        .
        .
        .
    LL:
    MM:
        IF A GR B
            THEN
                BEGIN
                A=A-B;
                COUNT=COUNT+1;
                GOTO MM;
                END
            ELSE
                GOTO OUT;
        .
        .
        .
    OUT:
        .
        .
        .
```

Figure 2-3. Label Statement Label Example

A labeled statement can be simply a labeled END. Although the following statement is valid for labeling an END for usage similar to the CONTINUE statement of FORTRAN, it is not particularly useful since SYMPL offers the TEST and RETURN statements which bypass the need for such labels:

```
FIN:END
```

Comments are valid after a label, as in:

```
NEXTONE: #CONTROL REACHES HERE TO GET
         THE NEXT CHARACTER#
```

STATEMENT FORMAT

SYMPL statements can be written anywhere within columns 1 through 72 on any number of cards or card images. Unlike FORTRAN and COMPASS, SYMPL attaches no significance to any particular column of a card. The compiler treats the source program simply as a stream of characters obtained from columns 1 through 72. Card boundaries are ignored.

Because SYMPL statements are format free, both of the examples in figure 2-4 are acceptable to the compiler. The first sequence not only allows the program logic to be followed more easily, it also allows easier modifications through utilities such as UPDATE or MODIFY. The conventions of the SYMPL coding form are recommended, in which labels appear in column 1 and declarations and executable statements begin in column 7, with only one statement appearing on a single card.

```
IF I LS MAX
THEN
    BEGIN
    A[I]=A[I] + 1 ;
    I=I+1;
    END
ELSE
    GOTO FIN;
NEXT:
    ENTER(SYMBOL,TABLE [I]);



    IF I LS MAX THEN BEGIN A[I]=A[I]+1;I=I+1;
    END ELSE GOTO FIN;NEXT:ENTER
        (SYMBOL,    TABLE
        [I] );
```

Figure 2-4. Statement Format Example

COMMENTS AND SPACES

A comment is written as a string of characters delimited by a pound sign, #. Comment character strings can contain any of the computer set characters except:

Pound sign

Semicolon

A null comment that consists only of two adjacent pound signs is legal also. Anyplace a space is legal, a comment is legal, with the exception of a DEF declaration or reference. In other than DEF:

A comment can substitute for a space.

A comment can be concatenated to any legal space.

The SYMPL metalanguage described in the SYMPL Reference Manual distinguishes between instances where a space or comment is required and instances where a space or comment can, but need not, appear. In general, a space or comment is required to separate reserved words and identifiers.

Instances in which spaces or comments are prohibited are:

Within reserved words: (GOTO not GO TO)

Within status constants: (S"RED" not S "RED ")

Before the colon in a label: (SUBR: not SUBR :)

Between P, B, or C and the left angle bracket of a P function or bead function: (C<2,6>ALPHA not C <2,6>ALPHA)

Spaces or comments are allowed before and after the following marks:

Semicolon that terminates statements

Comma that separates elements of a list

Arithmetic operators or unary operators

Colon in array dimensions

Brackets in array declarations

Parentheses enclosing formal parameters in a subprogram declaration

All the following statements are valid:

```
GOTO LAB;

GOTO LAB ;

GOTO##LAB;

GOTO LAB#ORDE#;

GOTO LAB # EXIT WHEN A=B # ;
```

A SYMPL program is a series of declarations and executable statements. It can be structured as a main program or as a subprogram. Since SYMPL is a systems programming language, most source code is written in the form of a subprogram rather than a main program. The two types of subprograms are procedures and functions; they differ in that:

A function returns a value through the function name. It is called when its name is used in an expression.

A procedure 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.

A program module is a separately compiled main program or subprogram. Compilation of a module is terminated whenever the compiler encounters a TERM statement.

If a subprogram is compiled in the same program module as the program or subprogram it is called by, it requires no special treatment. If a subprogram is compiled as a separate module, however, it is known as an external subprogram and any other module referencing it must acknowledge the external subprogram status.

Separately compiled programs and subprograms can communicate by any of the following ways, as described at the end of this section:

Declaring data in labeled or blank common

Declaring entities as external

Passing arguments in a procedure or function call

Passing parameters to a procedure using common instead of using formal parameters in the procedure call might improve execution speed. Differences in object program size vary depending on whether the program sets the common variables before each call, or with formal parameters, how many transfer vectors are recognized as duplicates.

General information about procedures and functions is contained in this section, along with information about alternative entries to these subprograms. Section 7 contains the details of parameters in subprogram declarations.

MAIN PROGRAMS

A main SYMPL program consists of a main program header, a single (usually compound) statement, and the ending reserved word TERM, as shown in figure 3-1. Notice that in SYMPL neither BEGIN nor END is followed by a semicolon.

Source statements between PRGM and TERM are compiled as a single relocatable binary module with a transfer address to the first executable statement.

A main program can include any number of embedded subprograms, and those subprograms also can include embedded subprograms. If TERM appears at the end of a subprogram, it stops compilation of the module in process and source statements following TERM are compiled as a separate module.

```
PRGM name;
    BEGIN
      .
      .
      .
    declarations and statements
      .
      .
      .
    END
TERM
```

Figure 3-1. Main Program Structure

The main program header establishes the program name:

PRGM name;

name Any identifier (1 through 12 letters, $, or digits not beginning with a digit) that is not a reserved word. For loader purposes, the name is truncated to seven characters.

PROCEDURES

A procedure is a subprogram that is called when its name is referenced. The procedure can, but need not, have an associated parameter list; also it can have alternative entry points. SYMPL procedures are similar to FORTRAN subroutines. They behave as PL/I or ALGOL procedures. They can be embedded within other procedures; nesting of embedded procedures is possible to any level.

SYMPL does not support recursive procedures; a procedure should neither call itself nor be called by any procedure it has called. Responsibility for avoiding recursion rests with the programmer. The SYMPL compiler, which does not have the stack mechanisms found in ALGOL and PL/I, does not check for recursion.

Procedures and functions can be nested, as shown in figure 3-2. In this figure the nested subprogram GEN has access to all data declared in the outer procedure MYSUB.

PROCEDURE DECLARATION

A declaration establishes a procedure. It can appear anywhere in a module, even after the procedure has been referenced. It is good programming practice, however, to group all procedure declarations together at the beginning of the program preceding any executable statements.

```
PROC MYSUB;
BEGIN
ARRAY T [100] ; ITEM TT;

    PROC GEN;
    BEGIN
          .
          .
          .
       TT [I] = 0;
    END # GEN #                    } GEN declared within MYSUB

    FUNC MAX(A,B) R;
    BEGIN
          .
          .
          .
    END # MAX #                    } MAX declared within MYSUB
.
.
.
GEN; } call to procedure GEN
.
.
.
X=MAX( Y, Z ) + MAX(V, W); } use of function MAX

GEN;
.
.
.
END # MYSUB #
```

Figure 3-2. Nested Subprograms

A procedure declaration can appear in either of the following formats:

procedure header
declarations for procedure
elementary or compound executable statement

or

procedure header
compound statement including declarations and executable statements

The usual form of a procedure includes all declarations and statements within the procedure header and the END which corresponds to the first BEGIN of the procedure. The name of the procedure is not required, but it can be included as a comment on the END statement. For example:

END#FINDIT#

The format of a procedure header is:

PROC name(param,param, . . .);

name — Any valid identifier that does not duplicate a reserved word.

param — Optional formal parameter used within the procedure for which an actual parameter is to be substituted at execution time.

A more thorough discussion of procedure declarations and parameters can be found in section 7.

A procedure declaration for a procedure SETIT is:

```
PROC SETIT (optional formal parameter list);
BEGIN
    .
    .
    .
END # SETIT #
```

The two declarations shown in figure 3-3 are legal and equivalent.

```
PROC P(A, B);
ITEM A,B;
BEGIN
ITEM I,J;
   .
   .
   .
END

PROC P(A, B);
BEGIN
ITEM A, B, I, J;
END
```

Figure 3-3. Procedure Declarations

Figure 3-4 shows a nested procedure REINITIALIZE within procedure MYSUB. Procedure REINITIALIZE can use any data of MYSUB without the need to pass that data formally. Notice that only the procedure name is used in the call; the four characters CALL do not precede the procedure name. For readability, however, many programmers use a DEF declaration (DEF CALL # #) to allow CALL in source listings.

```
PROC MYSUB;
BEGIN
ARRAY [100] ; ITEM A;
.
.
.
   PROC REINITIALIZE;
   BEGIN
   ITEM I;
   FOR I=0 STEP 1 UNTIL SIZE DO
      A[I]=0;
   FOR I=0 STEP 1 UNTIL LENGTH DO
      NAME[I]=" ";                 } procedure declaration
   END # REINITIALIZE #
.
.
.
REINITIALIZE;} call to procedure
.
.
.
REINITIALIZE;} call to procedure
.
.
.
END # MYSUB #
TERM
```

Figure 3-4. Procedure Declaration and Call Example

Notice that procedure REINITIALIZE executes when it is called, not when its declaration is encountered. When MYSUB in figure 3-4 executes, the statements of procedure REINITIALIZE are bypassed as though the program were written as shown in figure 3-5.

```
        .
        .
        .
        GOTO BYPASS;
            PROC REINITIALIZE;
                .
                .
                .
            END # REINITIALIZE #
BYPASS:
        .
        .
        .
```

Figure 3-5. Program Execution Flow

Procedures should be called only by the procedure name or alternative entry point name.

PROCEDURE EXIT

When control passes to a procedure, execution begins at the first executable statement associated with the entry point by which the procedure was called. Execution continues within the procedure until one of the following statements occurs:

END statement of the single procedure is reached and control returns to the statement following the procedure call.

RETURN statement within procedure is executed to return control to the calling subprogram.

STOP statement within procedure is executed to return control to the operating system.

GOTO statement is executed to transfer control to a label outside the procedure.

Exit from the middle of a procedure through a RETURN statement is illustrated in figure 3-6. Execution of procedure P occurs at its first call, after I has been assigned a value 0. After the first call, J has the value 0 since the RETURN statement executes. After the second call, which is entered with I=1, J has the value 1. A jump out of a procedure is valid, although good programming practices avoid such a jump.

In the example in figure 3-7, procedure JUMP has formal parameters N, L1, and L2. (The parentheses of N indicate call-by-value.) Since declarations for L1 and L2 do not appear within JUMP, the compiler considers them to be labels. The SWITCH declaration results in the compiler associating the identifiers of list MYSW with integer values 0 through 3, respectively. The IF statement sets N to one of these values; the GOTO statement jumps to the label associated with the value of N. The call to JUMP specifies the value of N and the particular label to be associated with switch values 0 and 3.

Section 7 describes parameters for procedure calls.

```
        .
        .
        .
        ITEM I, J;

            PROC P;
            BEGIN
            I=I + 1;
            IF I EQ 1 THEN RETURN;
            J=1;
            END # PROC P #

        I=0;
        J=0;
        P;
        P;
        .
        .
        .
```

Figure 3-6. Procedure Exit by RETURN Statement

```
    .
    .
    .
        PROC JUMP( (N), L1, L2 ) ;
        BEGIN
            SWITCH MYSW L1, LAB1, LAB2, L2;
            IF N LQ 0 THEN N=0;
            ELSE
                IF N GR 3 THEN N=3;
            GOTO MYSW[N] ;
        END # PROC JUMP #
        .
        .
        .
ERRMIN: . . .
LAB1: . . .
LAB2: . . .
ERRMAX: . . .
        JUMP(I, ERRMIN, ERRMAX); . . .
```

Figure 3-7. Procedure Exit by a Jump

FUNCTIONS

A function is a subprogram used within an expression. It returns a value through its function name. This value is then used in evaluation of the expression.

The two types of functions are:

Intrinsic functions that can be referenced at any time within a program without any FUNC declaration.

Programmer-supplied functions that must be declared within a program before they can be referenced.

INTRINSIC FUNCTIONS

The five intrinsic functions are:

ABS	Absolute function that obtains the absolute value of its argument.
B	Bit function that refers to bits in the specified item.

C Character function that refers to 6-bit characters in the specified item.

LOC Location function that obtains the address of its argument during execution.

P Pointer function that refers to the pointer to a based array.

The B, C, LOC, and P functions are described in section 5.

PROGRAMMER-SUPPLIED FUNCTIONS

Functions are similar to procedures in that they can be embedded within other subprograms, and they can be declared and referenced in suitably written separate modules. Parameter lists can be passed to functions. Alternative entry to a function can be declared within a function body, as described below. Recursive functions are not valid.

Functions differ from procedures in two respects:

A function must be declared before it is referenced.

A function declaration must contain an assignment statement that assigns a value to the function name.

Function Declaration

A function declaration must appear before the function is referenced in a module. It begins with a header followed by an optional series of declarations and a function body. The function body is a single elementary or a compound statement which can include the declarations as well as other elementary or compound statements. A statement assigning a value to the function name must be included in the function body.

The format of the function header is:

FUNC name (param,param, . . .) type;

name Any valid identifier that does not duplicate a reserved word.

param Optional formal parameter used within the function body for which an actual parameter is to be substituted at execution time.

type Type of result as described in section 4.

B	Boolean
I	Integer
U	Unsigned integer
S:stlist	Status
R	Real
C(lgth)	Character

If type is omitted, I is assumed.

Within the function body, the function name can be used only as the left-hand side of a replacement statement. In the example of a function declaration in figure 3-8, notice that the statement MAX=M sets the return value, thus fulfilling the requirement that a value be assigned to the function name. A real function MAX searches for the maximum value in array T[0:N]. Within the body, a statement such as IF TAB[I] GR MAX THEN. . . or MAX=MAX + 1 is invalid since SYMPL does not allow recursion.

```
FUNC MAX (T, (N)) R;
BEGIN
    ARRAY T; ITEM TAB R;
    ITEM N I, M I;
    M=TAB[0];
    FOR I=1 STEP 1 UNTIL N DO
        IF TAB[I] GR M
        THEN M=TAB[I];
    MAX=M;
END # FUNCTION MAX #
```

Figure 3-8. Function Declaration Example

A RETURN statement can appear in the function body to return control to the calling program, as long as the function name is assigned a value before RETURN executes. RETURN is not required to end a function.

Formal parameters within the function body are subject to the same scope of declaration rules as procedures.

Function Call

A function is called when its name appears in an expression. Each of the following statements is valid, assuming a prior declaration of function MAX having two formal parameters as shown in figure 3-8:

```
I=I + MAX (VECT, 17) ;

T[MAX (VECT, 17)]=K;

P(MAX (VECT, 17), X) ;
```

Actual parameters in the call must correspond to formal parameters in the function declaration. Parameters can be passed by value or address, as described in section 7 for procedure parameters.

Function calls compile as return jump instructions, with the result normally in register X6. When the result is data type C with a string of more than 10 characters, however, register X6 contains the address of the first word of a temporary storage area containing the string.

Real and integer functions are compatible with FORTRAN Extended 4; character value functions are compatible also, if the function is declared to be 10 characters or less. The function value is returned left-justified in a word, and the unused bits are not guaranteed to contain any specific value such as zeros or blanks.

ALTERNATIVE SUBPROGRAM ENTRY

Alternative entry points can be defined for both procedures and functions. The format of the declarations are, respectively:

```
ENTRY PROC name(optional formal parameter list);

ENTRY FUNC name(optional formal parameter
               list) type;
```

Entry names can be passed as parameters and declared as externals.

An example of alternative entry is shown in figure 3-9. Procedure INIT is declared with alternative entry INCREASE. When the procedure is called with INIT, array item TAB[I] is set to 0, but when called with INCREASE, array item TAB[I] is increased by 1.

As shown with procedure INIT in figure 3-9, the parameter list in an ENTRY declaration need not match the list in the subprogram declaration. If the same parameter name does appear in two entry points, it must be the same type of variable in each. A given parameter cannot be passed by value in one list and passed by address in another list. (See section 7 for parameter details.) PROC P1(A, (B)) and ENTRY PROC P2(A, B) are illegal, since B is not referenced identically in all entries to the procedure.

```
.
.
.
ARRAY TAB[0:100]; ITEM T;
INIT(TAB, 100);
INCREASE(TAB); . . .
INCREASE(TAB); . . .
INIT(TAB, 100);
.
.
.
     PROC INIT (A, (N)) ;
     BEGIN
          ARRAY A; ITEM AA;
          ITEM N, M, I, X;
          X=-1;
          ENTRY PROC INCREASE (A) ;
          X=X + 1 ;
          FOR I=0 STEP 1 UNTIL N DO
               AA[I]=X;
     END # INIT AND INCREASE #
```

Figure 3-9. Alternate Entry Example

COMMON BLOCK DECLARATIONS

Blank common and 509 labeled common blocks can be used to pass data to separately compiled programs and subprograms. The declaration for data in a given block must be the same in all program modules. The ITEM names can differ but the specifications must be the same.

To declare common storage, the format is:

COMMON name; data-declaration

name — Label for common block. Can be expressed as any legal identifier, but only the first seven characters become the block name.

If omitted, storage is allocated in blank common.

data-declaration — Scalar or array declaration. Can be expressed as a compound statement. If an array declaration is BASED ARRAY, only the pointer to the array is in common.

Data is never initialized in blank common. Data is initialized in labeled common only when one of the following conditions exists:

The program or subprogram is compiled with the P parameter on the SYMPL compiler call.

A CONTROL PRESET compiler-directing statement appears at the beginning of the program module.

Labeled common blocks are listed as part of the compiler output when either the X or R parameter is selected on the compiler call. The cross-reference map also lists labeled common names.

Good programming practices require use of meaningful names to improve readability when common is used. UPDATE common decks are particularly suitable for handling common. For example, assume the description of common block PARAMS is in an UPDATE common deck, as shown in figure 3-10. Decks that include P and Q should call the deck with PARAMS. The call to Q below has no actual parameters. Without the use of common, the declaration of Q would be Q(I1, I2, R3) and the call would take a form similar to Q(A[1],10,17.4). The use of XREF PROC Q within P is required because P and Q are separately compiled.

```
PROC P;
BEGIN
XREF PROC Q;
COMMON PARAMS BEGIN
                    ITEM I1, I2;
                    ITEM R3 R;
                    END
.
.
.
I1=A[1];
I2=10;
R3=17.4;
Q;
.
.
.
END # PROC P #
TERM

PROC Q;
BEGIN
COMMON PARAMS BEGIN
                    ITEM I1, I2;
                    ITEM R3 R;
                    END
.
.
.
FOR I=0 STEP I1 UNTIL I2 DO . . .
END # PROC Q #
TERM
```

Figure 3-10. COMMON Declaration Example

EXTERNAL DECLARATIONS AND REFERENCES

Any of the following SYMPL entities can be declared and referenced in separately compiled subprograms:

Scalar

Array

Based array

Label

Switch

Function

Procedure

The two SYMPL reserved words used for externals are:

XDEF Used in the declaring program to define the entity. This declaration generates an entry point that can be used by the loader.

XREF Used in the referencing program. This declaration generates an external reference to the entity. Use of XREF implies that the entity has been defined in another program.

XDEF and XREF are analogous to the COMPASS pseudo-instructions ENTRY and EXT, respectively. They are not analogous to FORTRAN Extended EXTERNAL statements.

DEFINING EXTERNALS

Storage is allocated for all entities declared with XDEF, just as if they did not have the XDEF designation. The declaration for an external definition of a scalar array, based array, or switch is simply the normal declaration preceded by the reserved word XDEF, as shown by the two examples in figure 3-11.

```
XDEF ITEM NAME C(7), MSGNUM I;

XDEF BEGIN
       ITEM NAME C(7), MSGNUM I;
       ARRAY [SIZE]; ITEM AA;
       SWITCH AUTOMAT DIGIT, LETTER, POINT,
           TEN, MARKS;
       END
```

Figure 3-11. XDEF Declaration Example

When a function, procedure, or label is declared to be external, however, the XDEF indicator is separate from the normal declaration. These three entities must be identified in two declarations:

The declaration appears in its normal format.

The XDEF indicator formats are:

```
XDEF PROC procname;

XDEF FUNC funcname;

XDEF LABEL labelname;
```

When more than one name is declared they must be contained in a compound statement. For example:

```
XDEF
BEGIN
    PROC PRGMA;
    PROC PRGMB;
END
```

The external declaration can appear anywhere within the scope of the corresponding name. XDEF is implicit in the outermost subprogram of a module and all its alternate entry points. Outermost entities should not be specifically declared external.

REFERENCING EXTERNALS

When a program references an entity that is defined and allocated storage in a separately compiled program, the referencing program must contain a declaration that states allocation exists elsewhere. No storage is allocated for an entity declared by XREF. The form of the declaration is affected by the kind of entity, but all such declarations begin with the reserved word XREF.

The declaration for a scalar, array, or label is simply the full declaration preceded by the word XREF, as shown by the two examples in figure 3-12. The declaration for a switch is XREF followed by SWITCH and the switch name, without the list of labels. For example:

```
XREF SWITCH ACTION;
```

```
XREF ITEM NAME C(7), MSGNUM I;

XREF BASED ARRAY SIZE;
    BEGIN
        ITEM LFN C(0,0,7);
        ITEM CS(0,41,18);
    .
    .
    .
    END
```

Figure 3-12. XREF Declaration Example

The declaration for a procedure is XREF followed by PROC and the procedure name. No parameters accompany the procedure name. For example:

```
XREF PROC Q;
```

The declaration for a function is XREF followed by FUNC and the function name and type. The function declaration must appear before the function is referenced. For example:

```
XREF FUNC SEARCH B;
IF SEARCH(FET) THEN GOTO ACTION[I];
```

The XREF declaration can take the form of a compound statement, as shown in figure 3-13. XDEF declarations can appear anywhere within the corresponding program. Except for procedures and labels, however, they must appear before they are referenced. All entities declared with XREF must have a corresponding entry point generated by an appropriate XDEF declaration or by being the outermost subprogram name in a module.

```
XREF BEGIN
       ITEM DATE C(8), TIME R;
       ARRAY FET;
       FUNC SUCC B;
       END
```

Figure 3-13. XREF Declaration as a Compound Statement

Data in a SYMPL program can be classified in terms of structure, type, or use.

Data structure is described by the terms scalar and array:

A scalar is a single element that occupies at least one full word of storage. A scalar is defined by an ITEM declaration.

An array is an arrangement of elements. An array is defined by an ARRAY declaration followed by either an ITEM declaration or a compound statement containing ITEM declarations. These ITEM declarations define elements within the array.

Data type is described by the terms integer, unsigned integer, real, status, character, and Boolean:

Integer, unsigned integer, and real data represent numbers in a form suitable for arithmetic. Such data is defined by a constant in an appropriate format or by an ITEM declaration with data type I, U, and R.

Status data is a variation of integer data in which the compiler substitutes integer values with names in a list. A STATUS declaration is required when data type S is specified in a declaration for a scalar or array item.

Character data is display code representation. Such data is defined by a constant in an appropriate format or by an ITEM declaration with data type C.

Boolean data can take on only the values TRUE and FALSE. Such data is defined by the constants TRUE and FALSE or by an ITEM declaration with data type B.

Data use is described by the terms arithmetic and Boolean:

Arithmetic data used in arithmetic expressions can be any type except Boolean.

Boolean data is used only in Boolean expressions. Boolean type is considered nonarithmetic.

CONSTANTS

SYMPL has five types of constants. Real, integer, status, and character data can be used in arithmetic expressions; Boolean constants can be used only in Boolean expressions. All types of constants can be used to preset a scalar or array item.

REAL CONSTANTS

Real constants are rarely used in system programming. They represent a numeric value containing a decimal point and are written in standard scientific notation with a string of decimal digits 0 through 9, a required decimal point, and optional sign. Optionally, a real constant can be written in exponential form with the characteristic and mantissa separated by the letter E. No embedded blanks are allowed. Examples of real constants are:

```
45.      0.0
98.9     6.4E+4
.4       31.415E-01
```

INTEGER CONSTANTS

Integer constants represent either a numeric value or a bit pattern. They can take the form of a decimal constant, an octal constant, or a hexadecimal constant.

The size of any integer constant is limited by the amount of storage allocated for it. Constants to be preset in an item are limited to item size. Only character type data can cross word boundaries. Constants used in expressions can be one word in size.

Decimal Constants

Decimal constants represent numeric values without a decimal point; a preceding sign is optional. They are expressed as:

decimal-integer

Decimal-integer must be a string of decimal digits 0 through 9, with no embedded blanks.

Examples of decimal integers are:

```
6        -24          4096
```

Octal Constants

Octal constants represent bit patterns, with each digit in the constant establishing 3 bits. They are expressed as:

O"octal-integer"

Octal-integer must be a string of octal digits 0 through 7; embedded blanks are ignored.

Examples of octal constants:

Octal Constant	Resulting Bit Pattern
O"777"	111111111
O"22"	010010

Hexadecimal Constants

Hexadecimal constants represent bit patterns, with each digit in the constant establishing 4 bits. They are expressed as:

X"hex-integer"

Hex-integer must be a string of 1 through 15 hexadecimal digits 0 through 9 and A through F; embedded blanks are ignored.

Examples of hexadecimal constants are:

Hexadecimal Constant	Resulting Bit Pattern
X"F"	1111
X"4BC"	010010111100

STATUS CONSTANTS

A status constant is a mnemonic for an integer that is set at compilation time by a STATUS declaration. A status constant has meaning only in conjunction with the STATUS declaration which contains the status name and status-values. A status constant is represented internally in the same way as a U data type item. Status constants are expressed as:

S"status-value"

Status-value is the name established by a STATUS declaration. Blanks are not permitted between S and the status-value; they cannot be embedded within a status-value.

Examples of status constants, assuming a STATUS COLOR RED,ORANGE,YELLOW declaration, are:

Status Constant	Value Compiled
S"RED"	0
S"YELLOW"	2

CHARACTER CONSTANTS

Character constants represent alphanumeric data with each character in the string representing 6-bit display code. They are expressed as:

"character-string"

Character-string must be a string of any characters from the computer character set. Maximum number of characters is 240. Any character " in the string must be expressed as " ".

Examples of character constants are:

"THIS IS A CHARACTER CONSTANT WITH
NON-SYMPL CHARACTERS ↑% "

"THIS ONE" "S TRICKY"

BOOLEAN CONSTANTS

Boolean constants represent the values TRUE and FALSE. They can be used only with Boolean expressions or items declared data type B. They are expressed as:

TRUE

FALSE

SCALAR DECLARATION

The ITEM declaration defines a scalar. SYMPL scalars are similar to FORTRAN variables, but they differ in several respects. In SYMPL:

Every scalar must be explicitly declared, including those used as DO loop variables.

The scalar declaration must appear before the first reference to the scalar.

No implicit characteristics are attached to scalar names.

Values can be preset in the scalar definition.

ITEM DECLARATION FORMAT FOR SCALARS

The format of a scalar declaration is:

ITEM name type = constant;

name — Any identifier of 1 through 12 letters, $, or digits beginning with a letter or $.

type — Data type; if omitted, I is assumed.

- I — Integer in which the leftmost bit is used for the sign and the remaining 59 bits represent the binary value. The compiler allocates a full word for an integer scalar.
- U — Unsigned integer in which all bits are used to represent the value. The compiler allocates a full word for an unsigned integer scalar.
- R — Real in which data appears in the single precision floating-point format standard for CYBER 170 systems.
- C(lgth) — Character in which data appears in standard 6-bit display code format with 10 characters to a word. The compiler allocates as many words as necessary for the character string; characters in the string are left-justified in the words. The character string length must be specified. It is a decimal integer constant of 1 through 240.

B — Boolean in which data appears as zeros or ones. The compiler allocates a full word for each Boolean scalar.

S:stlist — Status in which data appears as a small integer value assigned by the compiler from the positions of identifiers in list declared by a STATUS declaration. The compiler allocates a full word for each status scalar.

constant — Initial value of scalar to be preset at load time. Format of constant should be appropriate for the data type.

PRESET CONSTANT VALUES

A preset value can be assigned to a scalar at load time. The format of the constants are:

I or U — Integer constant in decimal, octal, or hexadecimal form.

R — Real constant with a decimal point.

C — Character constant in the form "string".

B — Boolean constant TRUE or FALSE.

S — Status constant in the form S"status-value".

Preset constant values are stored as presented by the constant form in the ITEM declaration, whether or not the constant agrees with the type specified. SYMPL neither converts nor checks for agreement between the type parameter and the preset value.

Constants preset by ITEM declarations are similar to those set by DATA statements in FORTRAN in that they are initial values only. During execution the value of a preset item can be changed, and once changed, it does not revert to its preset value even if the procedure that set it is called several times. For example, if the procedure shown in figure 4-1 is called three times, the output values of I are 1, 2, and 3, assuming OUTPUT is declared externally.

```
PROC P;
BEGIN
    ITEM I=0;
    I=I + 1;
    OUTPUT (I);
END # P #
```

Figure 4-1. Preset Constant Value Example

CONTRACTED ITEM DECLARATION FORMAT

A second ITEM declaration format allows more than one scalar to be declared.

ITEM name type=constant, name type=constant, . . . ;

Each name and type pair is independent of any other pair. Syntax is the same as described above.

EXAMPLES OF SCALAR DECLARATIONS

1. These examples show scalars without preset values:

   ITEM I;

   Integer scalar assumed for identifier I in the absence of a specified type parameter.

   ITEM OPERAND B;

   Boolean scalar.

   ITEM NAME C(7);

   Character scalar with string of 7 characters.

2. Scalars can be written in contracted form:

   ITEM I, OPERAND B, NAME C(7);

   Equivalent to example 1.

   ITEM A, B, C R;

   Equivalent to ITEM A I; ITEM B I; ITEM C R; It is not equivalent to ITEM A R; ITEM B R; ITEM C R;

3. Examples of scalars with preset values appropriate for the data type:

   ITEM NUM U=0;

   Unsigned integer scalar with 60 bits used as value.

   ITEM TOTAL=0;

   Integer scalar with rightmost 59 bits used as value and leftmost bit as + sign.

   ITEM FIRST B=TRUE;

   Boolean scalar with the value 1 in a 60-bit word.

   ITEM MESSAGE C(15)= "COMPILER ABORTS";

   Character scalar with COMPILER A in first word and BORTS with trailing blank fill in second word.

   ITEM MASK12 U=O"0101";

   Unsigned integer scalar creating bits 000001000001 at the rightmost end of the word.

4. Examples of scalars with preset values that do not correspond to the data type. Presets use the constant specified, even if it does not agree with the type declared:

   ITEM ONE R=1;

   Stored as 0. . . . .01, not normalized floating point format.

   ITEM SILLY I=FALSE;

   Stored as all 0 bits.

ARRAY DECLARATION

An array is an ordered set of entries defined by two consecutive declarations:

An array header that establishes the size and structure of the array.

A single ITEM declaration that describes the fields of the array. If more than one array item is declared, the declarations can appear between BEGIN and END.

Allocation of storage for an array depends on the array header:

An ARRAY declaration results in allocation of storage.

A BASED ARRAY declaration does not result in storage allocation for the array. Rather, it defines a structure that is to be superimposed over storage allocated elsewhere in the program and allocates one word to contain the pointer. Based arrays are described in section 5.

SYMPL arrays differ from FORTRAN arrays in several respects. In FORTRAN, an array has a name by which all elements in the array are known. Individual elements, which must be one word in length, are referenced by a subscript written in enclosing parentheses. FORTRAN numbers each dimension of the array starting with 1 and limits the number of dimensions to three.

In contrast to FORTRAN, an array in SYMPL need not have a name. Elements, which need not be all the same length and can contain more than one word, are referenced by their array item name, not the array name itself. Subscripts to an array item name are written in enclosing brackets. The number of dimensions in an array is limited to seven; each dimension can have a programmer-supplied upper bound and lower bound.

SYMPL offers many capabilities for array declaration that are not available in FORTRAN. These include:

Specifying bounds of a dimension with negative values, as in:

```
ARRAY [-10:-3];
```

Specifying array elements less than one word in size, as in:

```
ARRAY[4]; ITEM A C(5), B U(0,30,3);
```

Specifying array elements more than one word in size, as in:

```
ARRAY[4]; ITEM D C(46);
```

Presetting values in the array elements, as in:

```
ARRAY[4]; ITEM NUMS=[1,2,3,4,5];
```

Specifying storage structure for multiword elements, as in:

```
ARRAY[4] S(2);
```

Although arrays can have up to seven dimensions, system programming generally does not require multidimension structures. (The multiword element capabilities of SYMPL and its serial-versus-parallel storage structures allow results that might require more than one dimension in other languages.) Consequently, the following material deals mostly with single dimension arrays. See the SYMPL Reference Manual for a description of multidimensional arrays.

The complete format for an array declaration is shown here for reference only. Section 6 presents arrays in a tutorial manner.

```
ARRAY name [low:up,low:up,. . .] structure (esize);
```

name	Identifier naming the array.
low	Lower bound of a dimension of the array.
up	Upper bound of a dimension beginning at low.
structure	Structure of the array, P (parallel) or S (serial).
esize	Number of words required to hold one entry of the array.

SCOPE OF DECLARATIONS

An item declared within a subprogram is valid only within that subprogram and subprograms nested within it. Statements outside the declaring subprogram cannot reference that item by name. An item declared within a nested subprogram is valid only within that subprogram and any subprogram nested within it.

An item referenced only within the subprogram in which it is declared is called a local identifier. The compiler always allocates space for a local identifier. An item declared in one subprogram and referenced in a nested subprogram is called a global identifier.

Figure 4-2 illustrates local and global identifiers. In the procedure SUBPROG:

Identifiers for items D and E are local to procedure P. They are global identifiers for procedure R which is nested within procedure P. They are unknown outside procedure P.

Identifiers for items F, G, H, and I are local to procedure Q. They are unknown outside procedure Q.

Identifiers for items A, B, and C are local to procedure SUBPROG. They are valid anywhere in the body of SUBPROG. Therefore, A, B, and C are global identifiers for procedure P, procedure R, and procedure Q.

The statement D=A+B is valid within the body of procedure R. Within procedure Q, however, the same statement is invalid since D is unknown outside procedure P.

Procedure R cannot be called at any point marked #NO#.

If an item declared in a nested subprogram has the same name as a global identifier, the compiler allocates space for both identified scalars or arrays. The innermost declaration is valid only in the procedure in which it is declared. In case of conflict, the innermost declaration always has precedence. Consequently, two declarations can specify different types of data for the same identifier.

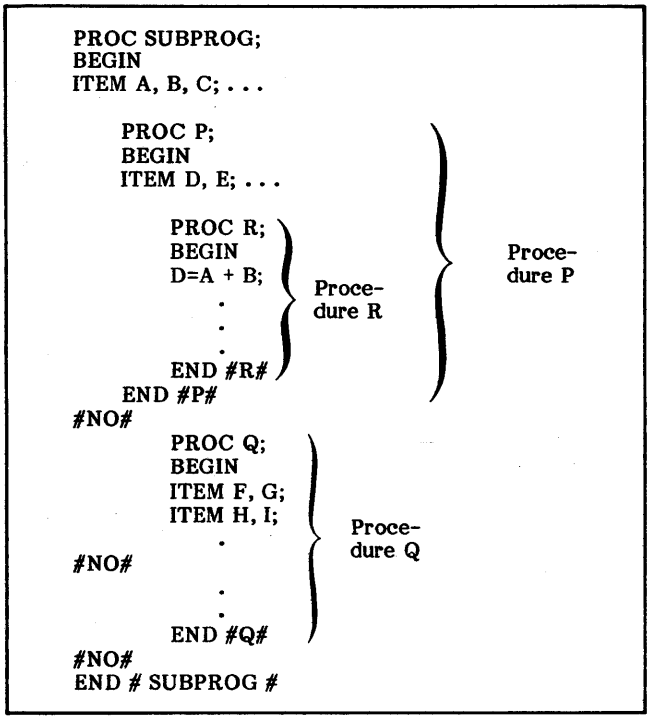


Figure 4-2. Local and Global Identifiers

In the example shown in figure 4-3, VAR is declared twice in procedure SUBPR. Since the innermost declaration has precedence, VAR in the body of procedure P is a local identifier of type Boolean and the statement VAR=TRUE is legal. The integer identifier VAR is valid anywhere in SUBPR except within procedure P. Since the two items, VAR, are in no way connected, good programming practice would be to give them unique names. In light of the systems nature of most SYMPL programs, it is also good programming practice to limit the use of global identifiers.

The subprogram in which an identifier is declared establishes the scope of declaration; the mere presence of a BEGIN . . . END sequence does not. BEGIN and END in compound statements such as FOR and IF do not affect the scope of an identifier.

```
PROC SUBPR;
BEGIN
ITEM VAR I;

    PROC P;
    BEGIN
    ITEM VAR B;
        .
        .
        .
    VAR=TRUE;
    END #P#
.
.
.
VAR=1;
.
.
.
END #SUBPR#
```

Figure 4-3. Duplicate Name Item Declarations

This section presents some of the declarations and statements that give SYMPL its power. These include:

DEF Declaration

References character strings by name for character string substitution during compilation.

SWITCH Declaration

Declares labels for a computed GOTO capability.

STATUS Declaration

Declares identifiers with implicit integer values for symbolic reference.

BASED ARRAY Declaration

Declares an array structure to be superimposed over data allocated elsewhere in program.

LOC Function

References addresses of other program entities.

Bead Functions

References part of a string of characters or bits.

DEF DECLARATION

The DEF declaration is a compiler-directing statement that allows a character string to be referenced symbolically in a program. The DEF declaration defines a name and a character-string to be substituted for subsequent occurrences of the name. The character-string, or DEF body, can be as simple as an integer constant for a DEF name used in array dimension syntax; or it can be a complete executable statement or part of such a statement.

The DEF declaration has two forms:

A DEF name without a formal parameter list resembles the COMPASS assembly language MICRO pseudo-instruction that allows symbolic reference to a character string (in SYMPL, micro delimiters are not used with micro references, however).

A DEF name accompanied by a formal parameter list resembles the COMPASS macro facility and FORTRAN statement function facility in which actual parameters are substituted for formal parameters when the DEF name is referenced.

During compilation, the character-string is substituted for occurrences of the DEF name. No computation takes place as a result of the substitution; a DEF name of ABC and a body of 3+2 results in the three characters 3+2 in place of ABC, not the single character 5.

Among the common uses of DEF are:

Improving program readability by allowing illegal words in the source listing. To allow a source statement such as IF A=0 THEN CALL ERROROUTINE:

```
DEF CALL # #;
```

Improving program execution speed by allowing in-line function code rather than the return jump execution of normal function calls:

```
DEF MODULO(X,N) #(X)-(X)/(N)*(N)#;
RANK=MODULO(LENGTH,10);
expands as RANK=(LENGTH)-(LENGTH)/(10)*(10);
```

Improving program maintainability by defining limit sizes that can be updated by future DEF changes:

```
DEF ENTRYLENGTH #3#;
ARRAY TBL[2] P(ENTRYLENGTH);
```

DEF cannot be used to redefine an identifier defined in an ITEM, ARRAY, or COMMON declaration:

```
ITEM ONE; DEF ONE #TWO#; produces ERROR nn
```

Further, DEF cannot be used to redefine the characters that serve to identify SYMPL syntax. Character substitution does not occur for the following characters used in the context of a syntax descriptor:

```
B, C, I, R, S, U, E, O, X, P
```

Substitution does occur when one of these characters is used as an identifier, nevertheless:

```
DEF C #NEW#;
ITEM ONE C(6); A=B + C;
expands as ITEM ONE C(6); A=B + NEW;
```

Similarly, DEF S #Q#; has no effect on a subsequent statement such as IF Z EQ S"SIZE". Substitution does not take place within a comment or within a character-string constant.

Any DEF declaration is effective only within the procedure or function in which it is declared, and it is effective only after the declaration appears. If a DEF name is redefined within a subprogram it does not affect the definition in an outer program. When the same DEF name is used again after leaving the subprogram, the DEF declaration in the outer program is effective.

DEF WITHOUT PARAMETERS

A DEF declaration without parameters produces straightforward expansion. The DEF format is:

```
DEF name #character-string#;
```

name — Any valid identifier by which the character-string is to be referenced.

character-string — DEF body that is to replace the DEF name during DEF expansion. From 1 to 240 characters can be used. The character # must appear as ##.

A space, but not a comment, can appear between the DEF name and the character-string.

The DEF body can contain another DEF name, as long as the definitions are not recursive or circular. The compiler checks for a single level of recursiveness only. Recursiveness obtained by nesting produces infinite loops.

DEF WITH PARAMETERS

A DEF declaration with parameters produces parameter name substitution during the expansion of the DEF body. The format for DEF with parameters is the same as without parameters, with the addition of a parameter list:

```
DEF name (param,param, . . .) #character-string#;
```

param — Formal parameter to be replaced by an actual parameter during DEF expansion. Must duplicate at least one identifier within the DEF body. If more than one parameter is used, they must be separated by commas.

If the number of actual parameters exceeds the number of formal parameters in the DEF declaration, a fatal error condition exists and expansion is suppressed. Expansion does occur, however, when the number of actual parameters is less than the number of formal parameters. The formal parameters without corresponding actual parameters are removed and nothing is inserted in those places. Debugging can be difficult when the number of formal parameters differs from the number of actual parameters.

During compilation, the formal parameter names within the DEF body are replaced by actual parameter names. For example, assume the following DEF declaration:

```
DEF RESET (A,N) #FOR I=0 STEP 1
    UNTIL N DO A[I]=0;#;
```

A reference RESET(T,64) produces:

```
FOR I=0 STEP 1 UNTIL 64 DO T[I]=0;
```

A DEF parameter is not recognized within a comment or a constant string. For example, assume the following DEF declaration:

```
DEF RESET (A,N) # ##SET A[2] TO "N" ## A[2]="N" #;
```

A reference RESET(T,64) produces:

```
#SET A[2] TO "N" # T[2]="N";
```

Expansion occurs in all other contexts except as a declaration name or within # or " pairs. For example, assume the following DEF declaration:

```
DEF PART (A,B,C,D) #C<A,B> C[D] = " "#;
```

A reference PART(W, X, Y, Z) produces the meaningless syntax:

```
Y<W,X> Y[Z] = " ";
```

Often, parentheses should be used within the DEF body to achieve correct results. For example, assume the following DEF declaration:

```
DEF MODULO(X,N) #X-X/N*N#;
```

A reference Y=3*MODULO(13+2, 6+2) produces:

```
Y=3*13+2-13-2/6+2*6+2; and subsequent evaluation as Y=42.
```

Yet DEF MODULO(X,N) #(X-(X)/(N)*(N))#; with the same reference produces:

```
Y=3*(13+2-(13+2)/(6+2)*(6+2)); and subsequent evaluation as Y=21.
```

Actual parameters must be separated by commas. The compiler recognizes parameters by balancing pairs of delimiters: ( ), < >, and [ ]. New parameters are not recognized within pairs of # delimiters, however. Consequently, any actual parameter that contains a comma, semicolon, right parenthesis, or an unbalanced (,), <,>, or [,], should be delimited by pairs of # marks.

The delimiting # are suppressed during expansion. For example:

All the following are valid as actual parameters:

```
F(L,N)

P((A+B)/C)

C<MODULO(N,64),J>T[K,L]

"T[K,L]"
```

Each of the following, however, must be within # pairs if they are to be used as actual parameters:

2,BYTPW	must be #2,BYTPW#
A=B; C=D;	must be #A=B; C=D;#
C<	must be #C<#

A comment can be passed as an actual parameter if the comment is enclosed by double # marks, as in:

```
DEF THREE(A,B,C) #A; B C; #
THREE(X=Y, ##SET Z TO X##, Z=X);
expands as X=Y; #SET Z TO X# Z=X;
```

A consecutive set of commas in an actual parameter string is valid to indicate an empty actual parameter, as in:

```
DEF THREE(A,B,C) #A; B C; #
THREE(X=Y, , Z=X);
expands as X=Y; Z=X;
```

SWITCH STATEMENT

A switch is a SYMPL concept that is similar to the computed GO TO statement of FORTRAN. The label to which control branches depends on the value of an expression at the time the GOTO executes. SYMPL has neither the assigned GO TO statement of FORTRAN nor the CASE statement of ALGOL.

The SWITCH declaration defines a named list of labels. The compiler associates the first label in the list with unsigned integer value 0, the second label is associated with 1, and so forth, through the list.

The SWITCH declaration is:

SWITCH swname label, label, . . . ;

swname — Identifier specifying the name of the switch.

label — Identifiers of labels to be associated with the list. Labels in the list need not have been previously declared. A label identifier can duplicate identifiers in other lists. Null positions in the list can be indicated by consecutive commas. Another switch name cannot appear in the list.

The switch name can be used only in a GOTO statement. It cannot be used in P functions or as a parameter for a function or procedure.

GOTO format is:

GOTO swname [arithmetic expression];

When GOTO executes, the expression is evaluated; control then transfers to the label whose value is equal to the value of the expression:

In the following, control transfers to label LDN when 3 is the value of I:

```
SWITCH DEVELOP TTO, ARH, SVL, LDN;

GOTO DEVELOP [I];
```

If evaluation of the expression in the GOTO statement produces a result that is beyond the values associated with the switch, execution results are unpredictable. Switch limit checking can be activated by the C parameter of the compiler call. When C is selected, an out-of-bounds reference results in a diagnostic message and execution aborts.

Within the SYMPL compiler, switches are implemented as sequential jumps. Normally, one element appears in each half of the word. Less space is consumed, but execution time is increased when switch packing is selected. Both the D parameter of the compiler call and the CONTROL PACK compiler-directing statement cause switches to be packed.

In the example in figure 5-1, the jump vector was compiled when neither the D nor the C option was selected for a declaration of SWITCH EVEN ZERO,,TWO,FOUR. With this declaration, an evaluation of I with a value of 1 creates an infinite loop.

```
EVEN    JP    ZERO
-       JP    ZERO
+       JP    *
-       JP    *
+       JP    TWO
-       JP    TWO
+       JP    FOUR
-       JP    FOUR
```

Figure 5-1. SWITCH Declaration Compilation

STATUS STATEMENT

STATUS is one of the more powerful concepts of SYMPL. The functions of STATUS can be duplicated by other programming techniques using integer values, but the simplification in programming, improvement in documentation, and advantages for program maintenance cannot be duplicated. STATUS is particularly useful in decision table and syntax analysis situations. Good programming practices call for use of STATUS whenever a set of variables is to be associated with small integer values.

STATUS is a compile-time concept similar to the EQU pseudo instruction of COMPASS. No memory is assigned to the status list mnemonics during execution.

The STATUS declaration defines a named list of mnemonics. The compiler associates the first mnemonic in the list with the unsigned integer value 0, the second mnemonic is associated with 1, and so forth, through the list. All items in the list always are referenced mnemonically.

The STATUS declaration format is:

STATUS stlist status-value, status-value, . . . ;

stlist — Name by which entire list is known, called a status list name.

status-value — Identifiers to be associated with the list. An identifier cannot be duplicated within a list. Unlike other program identifiers, however, they can duplicate the name of an identifier in any other status list or in the program, or even duplicate reserved word.

The following are equivalent:

```
ITEM A=3;

STATUS NUM  ZERO, ONE, TWO, THREE;
ITEM A S:NUM=S"THREE";
```

STATUS-VALUE REFERENCES

Status-values can be referenced in several forms, depending on the needs of a program:

A status function is used in all contexts in which the list and value must be associated.

A status constant is used in contexts in which the status list name is not ambiguous.

A status item provides convenience in referencing a scalar or array item that always takes status constant values.

A status switch is a SWITCH statement in which the switch name is associated with a status list and each label is associated with a status-value.

Status Function

A status function is actually a constant. It can be used anywhere in a program in which an integer constant can be specified, including array bounds specification and item presetting. Format is:

stlist "status-value"

During compilation the function is replaced by the appropriate value:

In the following statements, code generated during compilation presets item WHICH to the unsigned integer value 2 and assigns X=0:

```
STATUS KIND DOG, CAT, BIRD;
ITEM WHICH I=KIND"BIRD";
X=KIND"DOG";
```

Status Constant

A status constant is a shortened form for a status function. The format for a status constant, as defined in section 4, is:

S"status-value"

Because status-values are not required to be unique, the compiler must have some way to relate a status-value to the appropriate status list. This can be done by presetting the list name in an ITEM declaration, as in:

```
STATUS CLR RED, GREEN, GREY;
ITEM SHADE S:CLR;
IF SHADE EQ S"GREEN". . . .
```

where CLR is the list name and GREEN is the status-value being referenced.

A status constant can be used as loop control in FOR statements if the induction variable item has a status type. In expressions, the use of a status function or status constant is not restricted. If their meanings are not obvious, however, programming comments should be used extensively.

Status Item

If a scalar or array item usually contains a value from a particular status list, it should be defined as a status item. When this scalar or array is used in an expression with a status-value, a status constant can be used instead of a status function.

A status item is declared by a data type of:

S:stlist

Any place ITEM name U can appear in a declaration, the following can appear:

ITEM name S:stlist

Once an item of data type status is declared, status-values from the named list can be specified as status constants rather than the status functions that would otherwise be required. For example:

```
STATUS BULK ROBIN, OWL, EAGLE;
ITEM INCHES S:BULK;
ITEM WEIGHT U;
INCHES=S"OWL";
```

Without the status data type declaration for INCHES, the last statement must appear as:

```
INCHES=BULK"OWL";
```

With only the above declarations, INCHES=S"HAWK" produces a compilation error since HAWK is not a status-value from the list BULK.

Further, a statement such as WEIGHT=S"ROBIN" produces an error since WEIGHT is not a status item with ROBIN as a status-value.

A status item and status constant can be combined to preset an integer value. In the following example PAGE is set to 2:

```
STATUS SP NO, SGL, DBL, TPL;
ITEM PAGE S:SP=S"DBL";
```

Status items are not limited to status-values. Good programming practice, however, prohibits usage such as assigning a status-value to a status item for which it was not originally defined.

STATUS SWITCH

A status switch is a form of the SWITCH statement. Format is:

SWITCH swname:stlist label:status-value, label:status-value, . . . ;

swname	Switch name
stlist	Name of status list previously defined in a STATUS declaration
label	Name of label
status-value	Status-value from status list stlist that is to be associated with the preceding label

Figure 5-2 is an example of status switch use. Depending on whether NAME is alphabetic (has a display code less than 33), numeric (has a display code less than 45), or neither, NEXTCHAR is set to a certain status-value. At the end of their common processing, control transfers to the switch AUTO. If NAME is alphabetic, NEXTCHAR is set to status-value LETTER which is associated with label ALPHA, and control transfers to processing at label ALPHA. Control transfers in the same manner to NUMB if NAME is numeric, or to MARK if it is neither.

Label:status-value pairs can appear in any order. Not all status-values need be referenced in the switch. The same label can appear with more than one status-value. However, status values can appear only once. Figure 5-3 shows two examples of valid switch declarations.

EXAMPLES OF STATUS USE

The example in figure 5-4 declares status lists SOP and CLASS, which are sets of operators, with related status items initialized, respectively, to the last and first status values of the related list. The unnamed array has one element for mnemonic of status list SOP. Each array element is preset to a value that indicates whether it is an arithmetic operator, relative operator, or an error.

The following IF statement determines whether or not code at level EXP should be executed by comparing the value of a current element of CLASS with the value of a status constant acceptable for arithmetic operators:

```
IF CLASS [OP]=S "ARITH" THEN GOTO EXP;
```

```
STATUS CHAR LTR, DIGIT, OTHERS;
ITEM NEXTCHAR S:CHAR;
SWITCH AUTO:CHAR ALPHA:LTR,
                 NUMB:DIGIT,
                 MARK:OTHERS;
ITEM NAME;
          .
          .
          .
IF NAME LS 33
THEN NEXTCHAR=S"LTR";
ELSE IF NAME LS 45
    THEN NEXTCHAR=S"DIGIT";
    ELSE NEXTCHAR=S"OTHERS";
          .
          .
          .
GOTO AUTO[NEXTCHAR];
          .
          .
          .
ALPHA:
          .
          .
          .
NUMB:
          .
          .
          .
MARK:
          .
          .
          .
```

Figure 5-2. Status Switch Example

```
STATUS CHAR LTR, DIGIT, OTHERS;
SWITCH LOOP:CHAR LOOPA:DIGIT,
                 LOOPB:LTR,
                 LOOPB:OTHERS;

STATUS CHAR LTR, DIGIT, DIGIT2, OTHERS;
SWITCH LOOP:CHAR LOOPA:DIGIT,
                 LOOPB:DIGIT2,
                 LOOPC:OTHERS;
                 LOOPD:LTR;
```

Figure 5-3. Valid Status Switch Declarations

```
STATUS SOP PLUS, MINUS, EQ, LS, SOP;
STATUS CLASS BAD, ARITH, REL, COMP;
ITEM OP S:SOP=S"SOP";
ITEM KLASS S:CLASS=S"BAD";
ARRAY [SOP "SOP"];
ITEM CLASS S:CLASS = [S"ARITH",  # PLUS      1#
                      S"ARITH",  # MINUS     1#
                      S"REL",    # EQUAL     2#
                      S"REL",    #LESS THAN  2#
                      S"BAD"];   # ERROR     0#
```

Figure 5-4. Preset Status Values Example

Status constants can be used also as the loop control of a FOR statement, assuming an array TAB:

```
FOR OP=0 STEP 1 UNTIL S"SOP" DO
    TAB [CLASS [OP]]=TRUE;
```

Adding a new operator to the status list SOP in the example in figure 5-4 entails changing the STATUS declaration and adding a new element to CLASS:

```
STATUS SOP PLUS, MINUS, EQ, GR, LS, SOP;

S"REL", #GREATER THAN#
```

The IF and FOR statements in the example are not affected by the addition of the new operator. The addition was accomplished even though no empty array element was left for growth.

BASED ARRAY DECLARATION AND P FUNCTION

A based array is a structure for which no storage is allocated by the compiler. All references to items within a based array are compiled relative to the contents of its array pointer. The array pointer must be set explicitly within the program through use of the P function. Usually, the P function, for which one word is allocated, is assigned a value as the result of the LOC function reference to an array for which storage has been allocated.

In concept, a based array is a structure that can be superimposed over any portion of memory. By changing the pointer, the structure can be moved to various parts of memory. Based arrays in SYMPL (which have no similarities in COMPASS, FORTRAN, COBOL, or PL/I) provide flexibility for dealing with system programming concepts.

The declaration for a based array is the same as for a fixed array, except a name is required and preset values are not relevant:

BASED ARRAY name structure(esize); item description;

name	Required name of array.
structure	Indication of parallel or serial (P or S) structure of multiword entries. Default is P.
esize	Number of words in each entry. Default is 1.
item description	Description of entry in array, as described in section 4.

Several based arrays can be declared in a format:

```
BASED BEGIN ARRAY name . . . ;
            ARRAY name . . . ;
      END
```

The array dimensions can, but need not, be part of the BASED ARRAY declaration. If the subscripts of the based array and the array it is to be superimposed on need to be the same, the first element of a based array should correspond to the first element of the fixed array. SYMPL adjusts each array item reference at the time it is referenced, not at the time of the pointer setting. Accessing a based array item is slower than accessing a normal array item.

The pointer to the based array must be given a value through the P function. A P function is the name of the internal variable that contains the array pointer. It can be used the same as any variable.

The format is:

P<based array name> = arithmetic expression;

based array name	Name declared in BASED ARRAY declaration.
arithmetic expression	Arithmetic expression whose evaluation results in an address. Can be a LOC function, constant, or other expression.

On a word-addressable CYBER 70 or CYBER 170 system, any location in the program field length can be accessed as a subscripted word of a based array by:

```
BASED ARRAY ANY; ITEM X;
P<ANY>=0;
X[n]= . . .
```

The combined use of the BASED ARRAY declaration and the P function is illustrated in figure 5-5. The example assumes a file information table is allocated storage in procedure R. Procedure Q is to manipulate a file information table, with the array containing the file information table being passed as a parameter to Q. A reference to LFN[0]="MYFILE" in figure 5-5 sets the characters MYFILE in the first word of array FITNAME.

```
PROC Q(FITNAME);
BEGIN
    XREF ARRAY FITNAME;
    BASED ARRAY ALLFITS S(17);
        ITEM LFN C(0,0,7),
             RL  I(1,0,24),
             MRL I(6,0,24);
    .
    .
    .
    P<ALLFITS> = LOC(FITNAME);
    .
    .
    .
END
```

Figure 5-5. P Function Example

To superimpose the FIT structure on location 1000 octal in figure 5-5:

```
P<ALLFITS> = O"1000";
```

The P function can be used to represent the based array pointer variable in an expression. For example:

Assume the value of P<FIT> has been set to location 1000 octal. To move the based array structure 1000 octal words in memory:

```
P<FIT> = P<FIT> + O"1000";
```

A reference to LFN[0] in figure 5-5 then accesses location RA+2000 octal.

A based array can be used as a formal parameter in a procedure, though it is slow to access. The actual parameter must be a P function, not a based array name. This method is useful if the procedure is going to move the array.

For example, in figure 5-6, assume a based array A is to be used with the storage to which based array B currently points. Procedure P manipulates data known as array item. At the end of the procedure, the pointer to B must be reset to the pointer of A. Normally, the formal parameter is a fixed array, and the call passes a based array as an actual parameter.

```
PROC P(B);
BEGIN
BASED ARRAY B; ITEM . . . ;
BASED ARRAY A;
    BEGIN ITEM . . .

    END
P<A> = P<B>;
.
.
.
P<B> = P<A>;
RETURN;
END
```

Figure 5-6. Based Array as a Formal Parameter

A based array also is useful when a list is built dynamically in an area it shares with many kinds of data. For example, in figure 5-7a, assume a list in which STR points to a character string and SUC points to the next location in the string. Procedure ACTION manipulates each element of the list. The first parameter is the first element of the list; the second parameter is a procedure name (and consequently must be identified by FPRC). A call to procedure ACTION that would result in the printing of all elements of the list is shown in figure 5-7b. The subscripts can be omitted on STR and SUC because L has bounds 0:0.

a.
```
PROC ACTION ((FIRST),WHAT);
ITEM FIRST;
FPRC WHAT;
BEGIN
BASED ARRAY L;
    ITEM STR C(0,0,7), SUC I(0,42,18);
ITEM DUMY;
FOR DUMY = DUMY WHILE FIRST NQ 0 DO
    BEGIN
    P<L> = FIRST;
    WHAT (STR);
    FIRST=SUC;
    END
END #ACTION#
```

b.
```
ARRAY HEAP[1:200]; . . .
    PROC OUTPUT ((S));
        ITEM S C(7);
        BEGIN
        PRINT("(1H,A7)");
        LIST(S);
        ENDL;
    END #PROC OUTPUT#
ACTION(LOC(HEAP[N]),OUTPUT);
```

Figure 5-7. Use of a Based Array for Listing

LOC FUNCTION

LOC is an intrinsic function that returns the address of the actual argument used in the function call. The most common use of LOC is to obtain an address for a based array

pointer, but LOC is not restricted to such use. The value returned from the function is an address of type I.

The function call is:

LOC(argument)

argument Can be the name of any of the following:

Scalar
Subscripted array item
Procedure name
Function name
Label name
Switch name
Array name with optional subscript
P function

When LOC is used with the name of a based array as an argument, the value returned is the current value of the pointer, not the address of the pointer. When LOC is used with a P function, the address of the pointer word of the based array is returned. If the argument is an array item, the value returned is the address of the word where the item resides within the element.

For example, assume array ILFIT is a file information table declared in another module. It is accessed as a based array FIT with:

```
XREF ARRAY ILFIT; ITEM LFN C(0,0,10), . . . ;

P<FIT>=LOC (ILFIT);
```

In general, LOC should not be called with the name of a function, procedure, label, or switch, except perhaps during debugging. Although an address is returned, that address is probably not useful since no inferences can be drawn about the contents of locations surrounding the address returned. Further, the results from a particular program might not be reproducible when a different version of the compiler is used or a different optimization occurs with the same compiler version. For instance, with statements L:GOTO M; GOTO N; in a program, A = LOC(L) returns the address of label L, but A+1 does not reference GOTO N because the compiler can delete the statement and reorder the physical locations. Similarly, A = LOC(L+1) has no meaning, although the compiler does not prohibit such a statement.

One use of LOC is illustrated in figure 5-8. Assume a COMPASS main program with a 1000 word buffer at tag BUFFER. The SYMPL subprogram uses the buffer for writing, accessing the array as an XREF item. The buffer pointers are on array FET. After the first LOC function, FIRST points to BUFFER; after the second LOC function, IN points to the element BUFFER [CURRENT] which would be the last word of data written.

```
PROC WRITE;
BEGIN
XREF ARRAY BUFFER;
ITEM CURRENT;
ARRAY FET; ITEM FIRST . . . . IN . . . , . . . ;
.
.
.
FIRST [0]=LOC (BUFFER);
IN [0]=LOC (BUFFER [CURRENT]);
```

Figure 5-8. LOC Function Example

When file environment tables or other system interfaces are involved, SYMPL code cannot be used to monitor operating system activity. Optimization considers such data to be constant and might remove the tests from loops. See the SYMPL Reference Manual appendix C for more information.

BEAD FUNCTIONS

ITEM declarations define scalars, full words of arrays, or partial words of an array. Each time an identifier of an ITEM declaration is referenced, the entire contents of the item is accessed. At times, however, only part of an item is wanted. The bead functions (a bead is one of a string) provide access to part of an item for the purpose of extracting the contents of, or storing into, partial words.

In good programming practices, bead functions are used sparingly, since a program making frequent references to these functions is hard to maintain. Declaration and use of an array with partial-word items is preferable.

The two bead functions are:

C Access specified number of 6-bit bytes as data type character.

B Access specified number of bits as data type unsigned integer.

The two functions are not interchangeable; the C function implies the result is data type C, but the B function implies the result is data type U. The source data type is assumed to match the function, even if it is a different data type item. For example, the function B<42,18>, not the function C<7,3>, should be used to access an address in the lower 18 bits of an integer item.

Numbering conventions, which for the most part are not the conventions used elsewhere in the operating system, are as follows:

Characters and bits are numbered from 0, not from 1.

The leftmost bit is numbered 0.

The leftmost character is numbered 0.

Characters are each 6 bits.

If a bead function appears within a larger expression, SYMPL moves the specified item to a full word, aligning data as appropriate for its type. Then the result is used in the expression or replacement statement.

SYMPL does not check whether the number of beads to be extracted is within the size of the item. The programmer is responsible for the use of the function.

Bead functions can be used in the following circumstances:

In place of an item name in an expression, as in:

```
IF C<0,5> NAME EQ "INPUT" THEN R=1;
```

Left side of a replacement statement. Only the beads specified are affected, with any remaining characters untouched:

```
C<9,4> STRING=C5;
```

Right side of a replacement statement. SYMPL extracts the beads, then converts to the data type of the left side of the statement:

```
LFN=C <0,7> FITQ;
```

Parameters to a function or procedure. The function has the same properties as a subscripted variable in that it is computed and stored in temporary storage, and cannot be an output parameter:

```
CALLABC (J, C<9,1> NAME);
```

Bead functions can cross word boundaries only when the bead is extracted from a data type C item. Calls to library routines are compiled when a bead function crosses, or might cross, a word boundary, thus retarding processing. If the compiler can determine that only one word is to be accessed, the function is evaluated in-line. For example, given a data item LONG, in-line code results from:

```
C<12,3> LONG;
```

On the other hand, calls to library routines are compiled from:

```
C<I,J> LONG;
```

CHARACTER (BYTE) FUNCTION

The character function, which is also known as a byte function, extracts consecutive 6-bit characters from the specified item. The function is similar to the PL/I function SUBS. The result of a character function is data type C, with values assumed to be display code.

The format of the character function is:

```
C<start,number> identifier
```

start — Arithmetic expression indicating the first character to be extracted. Character positions are numbered from 0 at the left of the item.

number — Arithmetic expression indicating the number of consecutive characters to be accessed. The value of start+number should be within the size of the item.

If a length parameter is omitted, a single character is extracted.

If the data type of the item being accessed is C, the function can cross word boundaries and the maximum value for length is 240. (240 is the maximum number of characters allowed in a string.)

If the data type of the item is not C, however, the maximum value for length is 10.

identifier — Name of scalar or array item from which characters are to be extracted. Can be any data type, except B or S, but the result is always data type C. The extraction is done without any conversion.

EXAMPLES OF CHARACTER FUNCTION USE

1. To compare the hashed value associated with an identifier, the function shown in figure 5-9 adds the display code values of all identifier characters. The modulo 100 octal (decimal 64) is established through DEF so the subprogram could be easily modified for another modulo. The C function extracts one character at a time from the identifier. As with all functions, the name is set to the return value within the function.

```
FUNC HASH (IDENT) I;
BEGIN
DEF NCH #64#;
ITEM IDENT C(12);
ITEM I, H;
H=0;
FOR I=0 STEP 1 WHILE C<I,1> NQ " " DO
    H=H + C<I,1> IDENT;
HASH=(H)-(H)/(NCH)*(NCH);
END #HASH FUNCTION#
```

Figure 5-9. Use of C Function in a Hashing Routine

2. SYMPL limits character strings to 240 characters. Longer strings can be manipulated within a program as an array of strings. Procedure ADD, as shown in figure 5-10, adds up to 10 characters to the right end of a character string. The procedure has three parameters: the name of the array to which characters are to be added, the number of characters to be added, and the characters to be added. The first call to ADD moves three characters expressed as a constant; the second call uses a bead function to specify the location of characters to be moved.

   The example in figure 5-10 moves characters to a larger array BUFFER. The two DEF statements establish a byte number of a character within a word, and the word index of a character in a string buffer given its index I. I points to the first available character.

   The IF statement handles two conditions: the THEN clause adds characters when the characters to be added reside within a single word; the ELSE clause handles the situation when all characters are not in the same word.

   Good programming practice calls for a statement similar to DEF BYPW #10# with reference to the number of bytes per word referenced as BYPW. As stated above, the example is machine dependent.

3. An integer value between 0 and 9 can be converted to a decimal digit in display code by adding the character constant 0 to the integer. This is machine dependent, in that it depends on contiguous numbers in the character set.

   As shown in figure 5-11, function DECIMAL is a character function that converts N to a string of digits. Boolean item NEG is used to determine whether the leftmost character in working string STR is to be a minus sign. The absolute value function, ABS, is used with N prior to conversion.

```
ARRAY BUFFER [1000]; ITEM BUFWD C(10);
ITEM I;
DEF JB(I) #I-(I)/10*10#;
DEF JW(I) #I/10+1#;
ITEM LETTERS C(26)=
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
I=0;
ADD(BUFFER,3,"ABC");
ADD(BUFFER,10,C<3,10>LETTERS);
        .
        .
        .
    PROC ADD(SBUF,(NCHARS),(CHARS));
    BEGIN
    ARRAY SBUF; ITEM SBUFW C(10);
    ITEM NCHARS, CHARS C(10);
    ITEM I, L;
    XREF PROC ERROR;
    IF NCHARS LQ 0 OR NCHARS GR 10
    THEN
        ERROR ("ILLEGAL NCHARS");
    IF JB(I)+NCHARS LQ 10
    THEN
        C<JB(I),NCHARS>SBFW[JW(I)] =
            C<0,NCHARS>CHARS;
    ELSE
        BEGIN
        L=BYPW-JB(I);
        C<JB(I),L> SBFW[JW(I)] = C<0,L>CHARS;
        C<0,NCHARS-L> SBUFW[JW(I)+1] =
            C<L,NCHARS-L>CHARS;
        END
    I=I+NCHARS;
    END #ADD#
```

Figure 5-10. Use of C Function to Increase Character String Size

```
    FUNC DECIMAL((N)) C(20);
    BEGIN
    ITEM N;
    ITEM K, DIGIT;
    ITEM NEG B;
    ITEM STR C(20);
    STR=" ";
    IF N EQ 0 THEN BEGIN
                   C<19,1> STR="0";
                   RETURN;
                   END
    NEG=N LS 0;
    N=ABS(N);
    K=21;
    FOR DIGIT = N-N/10*10 WHILE N NQ 0 DO
        BEGIN
        K=K-1;
        C<K,1> STR = DIGIT + "0";
        N=N/10;
        END
    IF NEG THEN C<K-L,1> STR="-";
    DECIMAL=STR;
    END
```

Figure 5-11. Use of C Function for Number Conversion

BIT FUNCTION

The bit function extracts the specified number of consecutive bits from any specified item of type I, U, R, or C. The result of a bit function always is data type U, even if the bits are extracted from a different type of item.

A bit function cannot be used to obtain the absolute value of an integer. In the following example where INT contains a negative value, the result is probably a very large, positive number:

```
B<1,59>INT
```

The format of the bit function is:

```
B<start,number> identifier
```

start — Arithmetic expression indicating the first bit to be extracted. Bit positions are numbered from 0 at the left of the item.

number — Arithmetic expression indicating the number of consecutive bits to be accessed. The value of (start + number) should be within the length of the item. If a number parameter is omitted, a single bit is extracted. Number can be a maximum of 60.

If the data type of the item accessed is C, the function can cross word boundaries and the maximum value for (start + number) is 1440 (240 is the maximum number of characters allowed in a string). If the data type of the item is not C, however, the maximum value for (start + number) is 60.

identifier — Name of scalar or array item from which bits are to be extracted. Can be item of any data type except B or S, but the result is always data type U.

Bit functions extract beads from the item specified, not from the word in which the item is located. For instance, assuming a one-word array as shown in figure 5-12, B<29,2> BB[I] extracts bits 29 and 30 from item BB, which are bits 39 and 40 of the full word WW[I]. B<30,10> WW[I] is equivalent to XX[I].

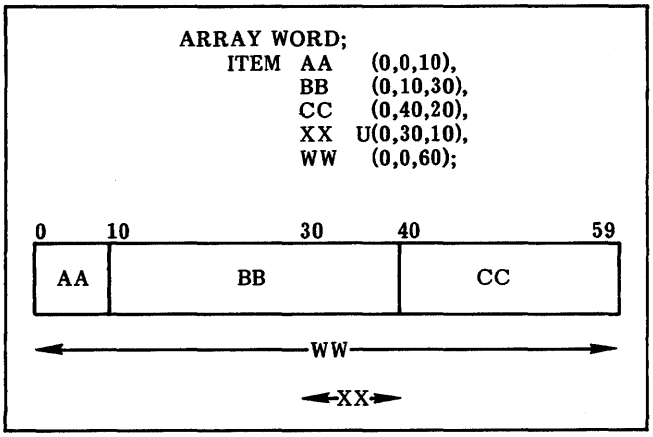


Figure 5-12. Bit Function Example A

EXAMPLES OF BIT FUNCTION USE

The first two examples show machine-dependent code.

1. An infinite operand if hardware compatible with CYBER 170 hardware (coefficient of 4000 or 3777 depending on a positive or negative operand) can be checked by:

```
DEF INF(X) # B<0,12>X EQ O"4000"
    OR B<0,12>X EQ O"3777" #;
```

2. An octal number in item N is converted to a string to be displayed as shown in figure 5-13.

```
FUNC DISPLAY ((N)) C(20);
BEGIN
ITEM N;
ITEM I, DIS C(20);
FOR I=0 STEP 1 UNTIL 19 DO
    C<I> DIS=B<I*3,3> N+"0";
DISPLAY=DIS;
END
```

Figure 5-13. Bit Function Example B

3. An integer variable from status list OPTION is used as a Boolean array as shown in figure 5-14. The item SET might be flag bits.

```
STATUS OPTION LIST, MAP, CROSS, TRACE;
ITEM SET;
DEF OFF # EQ 0 #;
DEF ON # EQ 1 #;
.
.
.
B<OPTION"TRACE"> SET=1;
.
.
.
IF B<OPTION"MAP"> SET ON AND
    B<OPTION"CROSS"> SET OFF THEN . . .
```

Figure 5-14. Bit Function Example C

4. Fields within an array item T are accessed by a subscripted array reference, even though the array is declared to have full-word items. The bead function defines three 20-bit fields in each array item. The example in figure 5-15 sets the pseudo-array item TT(10) to 0.

```
ARRAY [26]; ITEM T;
DEF TT(I) # B<( (I) - (I) / 3*3) * 20, 20> T[I/3] #;
ITEM A, B;
    A=6;
    B=4;
    TT (A + B)=0;
```

Figure 5-15. Bit Function Example D

MULTIWORD AND PART-WORD ARRAYS 6

An array is declared by an array header followed by an item declaration describing the named entity in that array. In section 4 the declaration was shown with the format:

```
ARRAY name[dimension bounds];

    ITEM name type;
```

For example, ARRAY[9]; ITEM ENTRY; defines an array of 10 entries, each one word in size. A reference to ENTRY [2] obtains the third word in the array. The total number of entries must be less than 65535. SYMPL arrays are not limited to one-word entries, however.

Assume an entry of 30 characters occupying 3 words. The array would be declared:

```
ARRAY [9] P (3);
    ITEM THREEWDS C(0,0,30);
```

A reference to THREEWDS[2] then obtains the third entry in the array, which is 30 characters long.

Suppose the entry has three words of related, but not identical, items. Rather than a 30-character string, an entry might consist of three separate items: a header word, an identifier name, and a pointer. The first and last words are data type integer, while the middle word is data type C. One means of describing this situation is through three arrays, but such declarations neither show the relationships between arrays nor offer the convenience of passing a single parameter to a subprogram. By using multiword array entries, the relationship can be maintained. To describe the three words suggested above:

```
ARRAY ALLTHREE[9] S(3);
    ITEM HEAD I(0),
         IDENT C(1,0,10),
         PTR I(2);
```

Subscripts (0), (1), and (2) determine the word in the entry in which the named item is to appear.

A reference to IDENT[9] picks out the second word of the last occurrence of the entry. Because the array is described with an S instead of a P, IDENT[9] occupies the next to the last word of storage allocated for that array. Use of ALLTHREE as a parameter to a procedure makes all occurrences of HEAD, IDENT, and PTR available to that procedure. The called procedure must have a formal parameter array with fields defined the same.

As an alternative to having more than one word, an entry can have less than one word. Consider a list of characters A through Z right-justified and zero-filled such as the FORTRAN compiler establishes for an array described by DIMENSION ALPHA(26) and DATA/ALPHA/. This array could be declared:

```
ARRAY[25] P(1);
    ITEM ALPHA C(0,54,1) =
        ["A", "B", "C", . . . "Z"];
```

A reference to ALPHA[10] obtains the 6-bit letter K, not 60 bits.

Further, one physical word of the array can have more than one item defined within it. Consider the 26 letters left-justified in the left-hand side of the word, with an address in the lower 18 bits of the word. The array with such an entry could be described:

```
ARRAY WHERE[25] P(1);
    ITEM ALPHA C(0,0,1),
         ADDR U(0,42,18);
```

Notice that the two item descriptions define only the leftmost 6 bits and the rightmost 18 bits of the word. Array item descriptions need not account for all bits in a word.

One further capability, overlapping, is possible in SYMPL arrays. A given bit in a word can be defined as part of more than one item. Suppose, in array WHERE above, the program needed to access the entire word, not just a part of a word. A third item, integer ALLOFIT, can be declared as shown in figure 6-1a.

Efficient overlapping of Boolean items can provide for testing many conditions with one elementary statement. In the example shown in figure 6-1b, testing B3 combines the tests on B1 and B2. B3 is true if either B1 or B2 is true. Testing is accomplished with one IF statement. The overlapping feature gives SYMPL capabilities achieved in FORTRAN by the EQUIVALENCE statement and in COMPASS by EQU.

a.
```
ARRAY WHERE[25] P(1);
    ITEM ALPHA C(0,0,1),
         ADDR U(0,47,18),
         ALLOFIT;
```

b.
```
ARRAY [10];
    ITEM B1 B(0,0,1),
         B2 B(0,1,1),
         B3 B(0,0,2);
    IF B3 THEN GOTO FINAL;
```

Figure 6-1. Item Overlapping

The only limits in combining multiword and part-word item descriptions are governed by physical word size:

All character data must be aligned at 6-bit boundaries.

Only items of data type C can cross word boundaries.

Items of data types other than C must be restricted to word boundaries.

COMPLETE ARRAY DECLARATION SYNTAX

The array discussion in section 4 and the examples above made use of abbreviated forms of array declarations. The

full array declaration, which allows multiword, fullword, or part-word entries to be combined in one of two storage formats is:

ARRAY name [low:up,low:up, . . .] st (esize);

name Identifier specifying the name of the array. It can be omitted unless the array is referenced in an XDEF, XREF, or BASED ARRAY declaration or a LOC function. No type is associated with the name.

low Lower bound of a dimension of the array. Must be expressed as an integer constant. Any value, including a negative value, can be specified, although a value of 0 offers execution efficiencies. If omitted, a value of 0 is assumed and the following colon must also be omitted.

up Upper bound of a dimension of the array. Must be expressed as an integer constant. Can be positive or negative. Must be equal to or greater than the preceding low with which it is paired.

st Structure of the array in storage:

S Serial in which all the words of one element are allocated contiguously.

P Parallel in which the first words of each entry are allocated contiguously, followed by the second word of each entry, and so forth.

esize Number of words required to hold one entry, expressed as an unsigned integer. If esize is omitted, 1 is assumed. esize must be less than 2048 words.

The ITEM declaration must immediately follow the ARRAY declaration. The format of the ITEM declaration for an array is:

ITEM name type(ep,fbit,size)=[preset],
name type(ep,fbit,size)=[preset], . . . ;

name Name of element in the array.

type Type of element: I, U, R, C(lgth), B, or S:stlist. If omitted, I is assumed.

ep Entry position. Word number within the element where the high-order bit of the item occurs, starting from 0. Must be expressed as an unsigned integer constant. If omitted, 0 is assumed.

fbit First bit. Beginning position of item within the word ep, counting from 0 on the left. Must be expressed as an unsigned integer constant. For a character item, character bit position 0,6,...,54. For other type items, bit number 0 through 59. If omitted, 0 is assumed.

size Item length expressed as an unsigned integer constant. For a character item, length is the number of characters not to exceed 240. For other type items, length is in bits not to exceed 60. R type data must have a size of 60. If size is omitted, 1 is assumed for Boolean and character, and 60 for all other data types. Only C type data can cross word boundaries.

preset Initial value for the item expressed as a series of values. The values must be arranged in the same order as the allocation of storage order and separated by commas. If omitted, no values are preset at load time.

If the entire field descriptor (ep,fbit,size) is omitted, defaults are as described above. If one parameter appears within the parentheses, it is assumed to be ep; two parameters are assumed to be ep and fbit.

The type indicator for an array item must specify the position the item occupies in the entry. For 60-bit items of type I, U, B, R, or S, position information can be abbreviated, but the word position is required. The three examples shown in figure 6-2 are all valid for an array declared as ARRAY LOOK[100] (3).

```
ITEM A    (0),
     B    B(1,0,60),
     C    C(2);


BEGIN
ITEM A    I(0,0);
ITEM B    B(1);
ITEM C    R(2,0,10);
END


ITEM B    B(1),
     C    C(2,0),
     A     (0,0,60);
```

Figure 6-2. Array Item Declarations

Notice that for data type C the format is not the same as for a scalar. For a 12-character item:

Scalar C(12)

Array item C(ep, fbit, 12)

An example of part-word items in an array with single-word entries is shown in figure 6-3.

```
ARRAY DESCRIPTOR S (1) ;
ITEM      N       B(0,0,1),
          V       B(0,1,1),
          A       B(0,2,1),
          KIND    U(0,27,5),
          TYPE    U(0,32,5),
          SCOPE   U(0,37,5),
          ADDR    I(0,42,18);
```

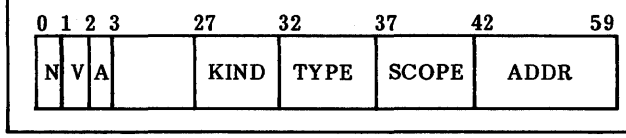


Figure 6-3. Array With Part-Word Items

When more than one item references the same field in the same word, the programmer is responsible for the results when data types are not alike. In the example in figure 6-4, INT[1] and CHAR[1] refer to the same word, but the types are different. Because of the differences in data type, different results are obtained from an assignment statement:

INT[1]=1 Sets right-justified integer 1.

CHAR[1]=1 Sets left-justified character A.

```
ARRAY CONST[24] S(3);
BEGIN  ITEM IDENT C(0,0,8);
       ITEM KIND  I(1);
       ITEM INT   I(2);
       ITEM CHAR  C(2,0,10);
END
```

Figure 6-4. Duplicate Field Item References

PARALLEL AND SERIAL ARRAYS

The capability to control array storage allocation is one of the outstanding features of SYMPL. For arrays with one-word entries, the distinction between P and S is meaningless. When multiword arrays exist, however, a parallel structure can decrease execution time.

For serial arrays, all words of the entry appear together. This is the normal structure for arrays such as the file name table in central memory resident or a FORTRAN double precision or complex array where entry size would be 2.

For parallel arrays, only entry words with the same entry position appear together. The structure can be visualized as an array of word [0] followed by an array of word [1], and so forth.

During execution, subscript calculations are faster for parallel arrays. Consequently, parallel arrays should be specified whenever possible. To access ONE[I], for instance, requires calculation of:

Parallel Address of ONE[0] + I

Serial Address of ONE[0] + 3 * I

Assuming an array SHOWIT with a 3-word entry containing three integer items, the different storage structures for serial and parallel allocation are shown in figure 6-5.

When an array item contains character data of more than 10 characters, the serial and parallel allocations still pertain to each 10-character word of the item within each word of the entry. Only items of data type C can cross word boundaries. Figure 6-6 compares serial and parallel allocation when multiword items are declared. For serial entry S, the two words of S[1] are contiguous. All entries with subscript [1] appear before any entries with subscript [2]. For parallel entry S, the two words of S[1] are not contiguous. All of entry S appears before any of entry T. The parallel structure is maintained within each entry.

PRESETTING ARRAYS

The first 6000 words of an array can have preset values. An array item is initialized by a string that contains one value for each occurrence of the item within the array. Presetting occurs for each item. The array declaration specifies what an item should be, not what a word should be. For example:

For a three-word array with one item in which occurrences are to be initialized to increasing integer values:

```
ARRAY LOOK [2];
   ITEM ONE=[0,1,2];
```

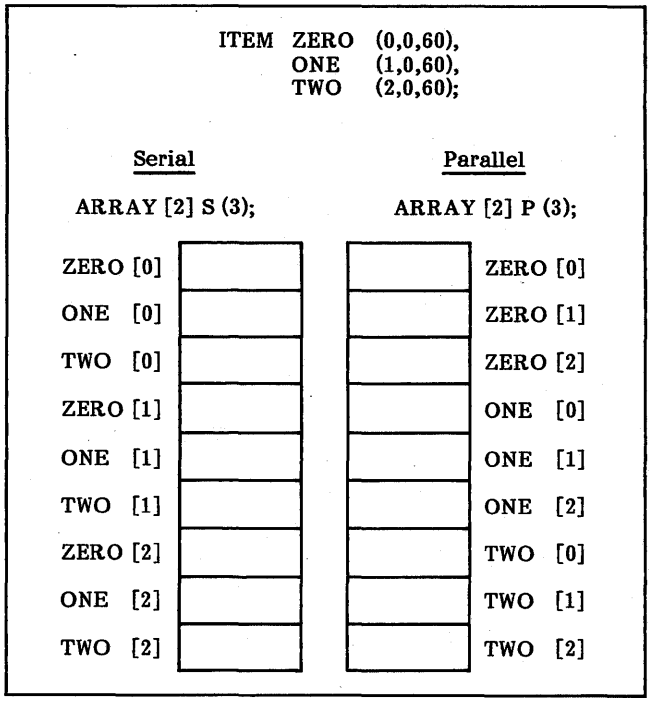


Figure 6-5. Serial and Parallel Allocation

A three-word array, SHOW, with two items in a single word, in which the occurrences of the first item are to be 1, 2, 3 and the second item A, B, C, is shown in figure 6-7. Bits 30 through 53 are undefined in the resulting array.

The string of values can be specified in abbreviated form, depending on the program needs:

If not all occurrences are to be initialized, a null value must be established by consecutive commas.

Trailing commas can be omitted.

If all occurrences are to be preset to the same value, an abbreviated format can be specified. To set ONE to all 0, for example:

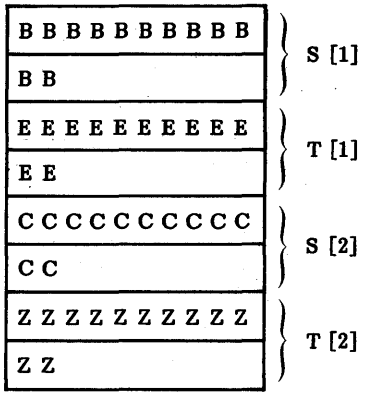
```
ONE=[3(0)]
```

An example of array presetting when not all occurrences are to be initialized is shown in figure 6-8. The elements without preset values are undefined, not zero. An example of array presetting with character data when not all occurrences are to be initialized is shown in figure 6-9.

If a number of occurrences are not to be initialized, an abbreviated format can be specified. For example, [10( ),1] is equivalent to [,,,,,,,,,,1].

Serial

```
ARRAY  SER[1:2] S (4);
ITEM   S C(0,0,12)=["BBBBBBBBBBBB",
           "CCCCCCCCCCCC"],
       T C(2,0,12)=["EEEEEEEEEEEE",
           "ZZZZZZZZZZZZ"];
```



Parallel

```
ARRAY  PAR [1:2] P (4);
ITEM   S C(0,0,12)=["BBBBBBBBBBBB",
           "CCCCCCCCCCCC"],
       T C (2,0,12)=["EEEEEEEEEEEE",
           "ZZZZZZZZZZZZ"];
```

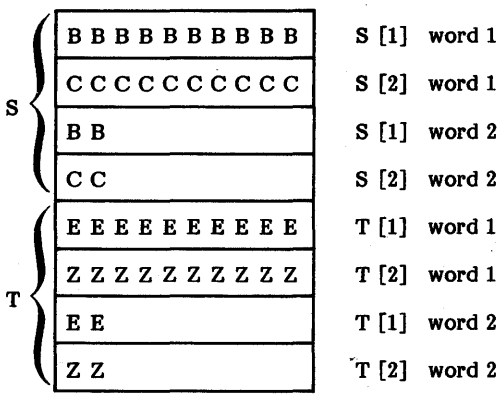


Figure 6-6. Serial and Parallel Allocation of Multiword Items

Presetting occurs without any check of array size. As a result, overlapping of values can occur. No error occurs if array bounds are exceeded. In the example in figure 6-10a, item X is initialized here only because of its position in relation to T. The compiler does not guarantee that array X immediately follows array T unless these declarations are within a COMMON block.

In the example in figure 6-10b, the preset values for BB, and then CC, overlap the preset values for AA. DD is initialized to the preset values of CC. This order is not guaranteed unless these declarations are in COMMON.

```
ARRAY SHOW[2];
ITEM FIRST (0,0,30)=[1,2,3],
     SECOND C(0,54)=["A", "B", "C"];
```

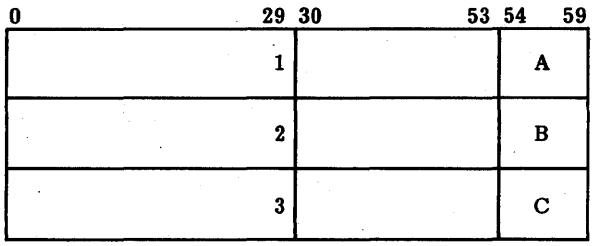


Figure 6-7. Array Presetting Example A

```
ARRAY [10];
ITEM T=[,2(1,,2),,,3];
```

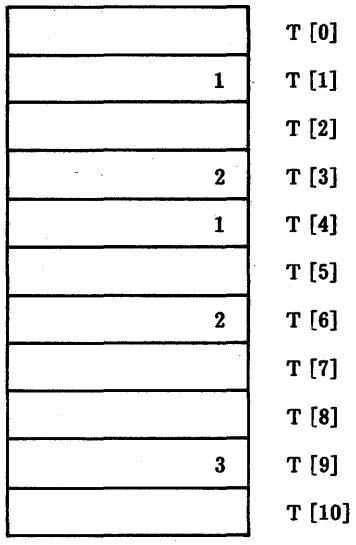


Figure 6-8. Array Presetting Example B

```
ARRAY [3] S (2);
ITEM H C(0,0,12) = ["T", ,"EEEEEEEEEEEE"];
```

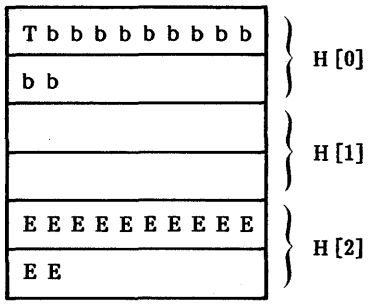


Figure 6-9. Array Presetting Example C

a. ARRAY [1]; ITEM T=[1,2,3,4,5];
ARRAY [4]; ITEM X;

1	T [0]
2	T [1]
3	X [0]
4	X [1]
5	X [2]
	X [3]
	X [4]

b. ARRAY A;
ITEM AA=[1,2,3,4,5];
ARRAY B;
ITEM BB=[6,7,8,9,10];
ITEM CC=11;
ARRAY D [1:3];
ITEM DD;

1	AA [0]
6	BB [0]
11	CC
8	DD [1]
9	DD [2]
10	DD [3]

Figure 6-10. Array Presetting Example D

Arrays can be initialized through status constants as well as the more common integer, Boolean, or character constants. Assume an array with two one-word entries as shown in figure 6-11. The second entry is associated with a status value from status list KIND.

Presetting of items that occupy only part of a word is the same as for whole word or multiword items. The setting of arrays TENSER and TENPAR in serial and parallel structures, respectively, is shown in figure 6-12.

PART-WORD ITEM EFFICIENCY

When an array with part-word items is being designed, efficiencies in access can be planned. Although SYMPL allows fields to occupy almost any position in an entry, good programming practice favors certain constructions for data of certain types.

When two items in different arrays are frequently exchanged, execution proceeds more quickly when the items occupy the same position within a word.

BOOLEAN DATA

The most efficient length for Boolean data is one bit. When a Boolean item is one bit in length, a shift to bit 0 and a sign

```
STATUS KIND NUMB, INT, REAL, BOOL, CHAR;
ARRAY STAN [4] S(2);
BEGIN
    ITEM IDENT C(0,0,10)=
        ["SIN",,,"SUBS", "ABS"];
    ITEM SKIND S:KIND(1)=
        [S"REAL",,,S"CHAR", S"NUMB"];
END
```

S I N	IDENT [0]
2	SKIND [0]
	IDENT [1]
	SKIND [1]
	IDENT [2]
	SKIND [2]
S U B S	IDENT [3]
4	SKIND [3]
A B S	IDENT [4]
0	SKIND [4]

Figure 6-11. Array Presetting Example E

test are the only instructions needed to determine whether it is TRUE or FALSE. If the item is more than one bit, however, the field must be masked and tested for a value other than 0.

An exception exists when one item overlays two others. In the following, item B1ANDB2 can be used to test for B1 or B2:

```
ITEM B1 B(0,18,1),
     B2 B(0,19,1),
     B1ANDB2 B(0,18,2);
```

Boolean data is most efficient in bit 0. No shifts or masks are required to access it.

Efficiencies in decision tables can be achieved when Boolean values are packed within a single word, as shown in figure 6-13. The STATUS function assigns integer values 0 through 3 to FO. The dimensions of array CHARACTERS are assigned through status functions and are [0,3]. To check whether a delete operation is valid, the following IF statement can be used:

```
IF VALIDDELETE [FO] THEN ZAPIT;
    ELSE ERROR;
```

INTEGER DATA

Signed integer data (declared by data type I) can be accessed more quickly on CYBER 170 compatible systems when the field is 60 bits or 18 bits and the field begins in bit 0 or the field occupies bits 42-59. Signed integers are faster than unsigned integers. Unsigned integers (declared by data type U) are accessed more quickly when the field ends with bit 59.

Serial allocation:

```
ARRAY TENSER[4] S (2);
BEGIN
ITEM A I(0,0,30)=[4,,3,,6];
ITEM B I(0,30,15)=[,3,,7];
ITEM C C(1,0,5)=["LLLLL", "BBBBB", "CCCCC",
     "TTTTT", "EEEEE"];
END
```

0 29	30 44	45 59	
4			A [0] , B [0]
L L L L L			C [0]
	3		A [1] , B [1]
B B B B B			C [1]
3			A [2] , B [2]
C C C C C			C [2]
	7		A [3] , B [3]
T T T T T			C [3]
6			A [4] , B [4]
E E E E E			C [4]

Parallel allocation:

```
ARRAY TENPAR[4] P(2);
BEGIN
ITEM A I(0,0,30)=[4,,3,,6];
ITEM B I(0,30,15)=[,3,,7];
ITEM C C(1,0,5)=["LLLLL", "BBBBB", "CCCCC",
     "TTTTT", "EEEEE"];
END
```

0 29	30 44	45 59	
4			A [0] , B [0]
	3		A [1] , B [1]
3			A [2] , B [2]
	7		A [3] , B [3]
6			A [4] , B [4]
L L L L L			C [0]
B B B B B			C [1]
C C C C C			C [2]
T T T T T			C [3]
E E E E E			C [4]

Figure 6-12. Array Presetting Example F

```
STATUS FO SQ, WA, IS, AK;
ARRAY CHARACTERS [FO"SQ":FO"AK"];
    BEGIN ITEM  D1 B(0,0,1) = FALSE,
                D2 B(0,1,1) = FALSE,
                D3 B(0,2,1) = TRUE,
                D4 B(0,3,1) = TRUE,
    VALIDDELETE  B(0,0,4);
    END
```

Figure 6-13. Packed Boolean Array

ACCESSING ARRAY ITEMS

A particular occurrence of an array item is referenced by:

item-name [subscript]

subscript — Arithmetic expression indicating occurrence of item. Can be signed integer constant, an unsigned integer constant, a scalar, array item, or an expression that provides such a value.

Both of the following are valid:

TAB [I + TAB[3] * 2]

X [-3] where ARRAY [-5:5]; ITEM X;

SYMPL does not check array bounds during execution. If the subscript is omitted from a reference to an array item, 0 is assumed. A diagnostic is generated unless the array bounds are 0:0.

When multiword items are referenced, the format is the same. The compiler generates code needed to extract the item from its serial or parallel array. Code generated for parallel arrays is more efficient.

When a part-word item is referenced, the compiler generates code that:

Masks the item to extract it from its word.

Shifts it to the position appropriate for its data type. Character data is left-justified and signed integer data is right-justified with sign extension, as described in section 4.

Only the referenced item is affected by access. No other part of the word in which the item is positioned is disturbed.

The usual form of a procedure declaration is shown in figure 7-1. The formal list of parameters identifies parameters to be passed to the procedure when the procedure is called by a reference to its name.

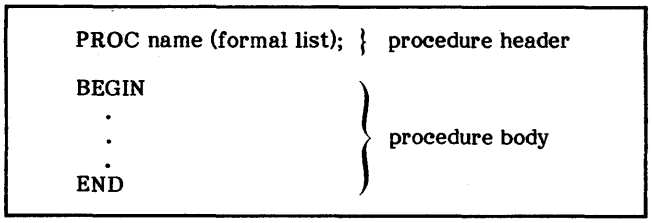


Figure 7-1. Procedure Declaration Structure

The procedure declaration establishes formal parameters that are used within the procedure body declaration. At the time the procedure is to execute, the actual parameters accompanying the procedure reference take the place of the formal parameters. The programmer is responsible for correspondence between the formal parameters and actual parameters. SYMPL checks neither the number nor type of parameters on a call during compilation or execution.

Formal and actual parameters are illustrated in figure 7-2a. Assume procedure SUB. The header for nested procedure P defines three formal parameters: A, BOUND, and S. Parameters A and BOUND are used within P to check whether the array bound is positive and to initialize the array items to a zero value. Parameter S returns a character constant to the calling procedure SUB. The header specifies that parameters A and S are to be passed by their address, but BOUND is to be passed by value.

When procedure P is called, actual parameters TAB, 64, and ETAT are substituted, respectively, for formal parameters A, BOUND, and S. Procedure P executes as if the lines containing the comment * were written as shown in figure 7-2b.

```
a.  PROC SUB;
    BEGIN
        PROC P(A,(BOUND),S);
        BEGIN
        ARRAY A; ITEM AA;
        ITEM BOUND, S C(10);
        ITEM I;
        XREF PROC ERROR;
    #*# IF BOUND LS 0
    #*# THEN ERROR("BOUND NEGATIVE");
    #*# FOR I=0 STEP 1 UNTIL BOUND DO
            AA[I]=0;
    #*# S="INIT";
        END # PROC P #
    ITEM ETAT C(10);
    ARRAY TAB[64]; ITEM T;
    P(TAB, 64, ETAT);
        .
        .
        .
    END # PROC SUB #


b.  BOUND=64;
    IF BOUND LS 0
        THEN ERROR ("BOUND NEGATIVE");
    FOR I=0 STEP 1 UNTIL BOUND DO
        T[I]=0;
    ETAT="INIT";
```

Figure 7-2. Formal and Actual Parameters Example

PROCEDURE DECLARATION AND CALL

The format for a procedure header with formal parameters is:

PROC name (param1, param2, . . .);

name Any SYMPL identifier (1 through 12 letters, $, or digits beginning with a letter) that is not a reserved word. Names of intrinsic functions are not reserved words. If the procedure is to be called by a program written in a language other than SYMPL, only the first seven characters are used and the first character cannot be the dollar sign.

param Name of any SYMPL entities listed below. Later in this section, each type of parameter is discussed separately.

Array	Based array
Function	Procedure
Label	Item

If the name specifies an item, it can be enclosed in parentheses to indicate a call-by-value rather than a call-by-address.

Within the procedure body, all formal parameters of any type except label must be declared. Any formal parameter not declared in the body is assumed to be a label parameter.

SYMPL makes certain assumptions about the formal parameters, depending on their type, as shown in table 7-1. Again, depending on the type, table 7-2 lists reasonable entities to pass as actual parameters. Notice that an array or based array formal parameter can be passed in several forms. Switch elements cannot be passed as parameters. (A switch can be used as an external entity, however.)

TABLE 7-1. FORMAL PARAMETER ASSUMPTIONS

Formal Parameter Declaration	Assumed Content of Parameter Word
ITEM I	Address of item I
ARRAY A	First word address of array A
BASED ARRAY BA	Address of pointer to BA
LABEL L	Address of label L
FPRC FP	Address of entry to procedure FP
FUNC F	Address of entry to function F

TABLE 7-2. POSSIBLE ACTUAL PARAMETERS

Formal Parameter	Reasonable Actual Parameter
Item for call-by-address	Item name, (item name)
Item for call-by-value	Item name, arithmetic expression, subscripted array item
Array	Array name, based array name
Based array	P function, item name, expression whose result is a pointer
Label	Label name
Procedure	Procedure name
Function	Function name

SCALAR AND ARRAY ITEM NAMES AS PARAMETERS

Any scalar or array item of any type (I, U, R, C, S, or B) can be specified as a formal parameter. Within the procedure, the formal declaration syntax is the same as a declaration outside the procedure.

The formal parameter list indicates whether a given parameter is to be passed by value or address. This is illustrated in figure 7-3. The corresponding actual parameter for W or Z is assumed to be a local variable with the address of W or Z; for X or Y, the assumption is that they are the actual values to be used for X or Y. Call-by-address is required for any parameter whose value is to be returned to the calling subprogram since call-by-value parameters work with a temporary copy of a variable.

An actual parameter can be a scalar name, constant, array item name, or expression. (When an actual parameter is the name of an item enclosed in parentheses, SYMPL considers it to be an expression.) Consequently, the procedure receives the address of a temporary location containing the

```
PROC P(W, (X), (Y), Z);
BEGIN ITEM  W R,
            X I,
            Y C(14),
            Z B ;
.
.
.
END #P#
```

Figure 7-3. Passing Parameters by Value or Address

scalar value instead of the address of the scalar itself. Such a parameter does not create the instruction savings of a call-by-value parameter. It does, however, provide the protection for the scalar value accorded all call-by-value parameters.

SYMPL performs no conversions when the type of a formal parameter is not the same as the type of the actual parameter:

In the following, X=FALSE at the end of the procedure since 1.0 is type real and has a format that is not the same as integer type 1:

```
R(1.0, 1, X);
    PROC R( (A), (B), C );
    ITEM A B,B B,C B;
    C = A EQ B;
```

No BEGIN and END pair is associated with procedure R. The declarations for a procedure can appear before the executable statement. Since C=A EQ B constitutes the entire executable portion of R, a compound statement is not required. Many of the remaining examples in this section use such a single elementary statement in the procedure called.

EXPRESSIONS AS ACTUAL PARAMETERS

Expressions are evaluated and the result is passed to the procedure as a temporary storage word. The procedure receives the address of a temporary location containing the result of the evaluation.

In figure 7-4, when procedure P is called, the statement changing W in the procedure has no effect. The temporary storage word for A+B is changed.

```
ITEM A=1, B=2;
P(A+B, B, 3);
      .
      .
      .
    PROC P(W, (X), (Y));
    BEGIN ITEM W, X, Y;
    ITEM I;
    X=X+1;
    FOR I=0 STEP 1 UNTIL X DO
    W=W+Y;
    END # PROC P #
```

Figure 7-4. Expression as Parameter

SUBSCRIPTED VARIABLES AS ACTUAL PARAMETERS

Subscripted variables are considered to be expressions. The procedure receives the address of a temporary location containing the result. As with parameters called by value, subscripted variable parameters modified within the procedure cannot be passed out of the procedure. For example, assume procedure P is defined by:

```
PROC P(A);
ITEM A;
A = 0;
```

When procedure P is called from a program containing the following statements, T[12]=2 when the calling program resumes execution:

```
ARRAY TT; ITEM T;
T[12]=2;
P(T[12]);
```

If a procedure must modify a subscripted variable, the array name and the subscript must be passed as separate parameters. In the formal array, the variable to be modified must be described as the same field as the actual item in the actual array. For example, assume a procedure Q defined, as shown in figure 7-5. When procedure Q is called from a program containing the following statements, T[12]=0 at the end of the procedure. Items X and T must have identical field descriptions:

```
ARRAY TT[100]; ITEM T;
T[12]=2;
Q(TT,12);
```

```
PROC Q(XX,Y);
BEGIN
ARRAY XX; ITEM X;
ITEM Y;
X[Y]=0;
END
```

Figure 7-5. Subscripted Variable as Parameter

CHARACTER STRINGS AS PARAMETERS

Character strings are passed to a procedure without any accompanying information about length. The programmer writing the procedure is responsible for knowing the length.

The declared length of the string cannot be passed to the procedure through a variable in the parameter list. ITEM STRING C(N) is illegal in a procedure since the syntax of an ITEM declaration requires a character string length to be expressed as an integer constant. The compiler generates code based on the declared length of the formal parameter. If the actual parameter is not the same length, unexpected results can occur.

The actual parameter string should have a length longer than or equal to the formal parameter string length. If it is longer, only the number of characters specified by the ITEM declaration are used or altered. An actual parameter string shorter than the formal parameter string can produce unpredictable results, since characters following the actual parameter are accessed. The compiler does not guarantee the contents of those characters.

No padding occurs when an actual parameter string is shorter than a formal parameter string. In the example in figure 7-6, the call to procedure Q sets the first 10 characters of LEFT to the value RIGHT; the last 10 characters are undefined.

```
ITEM LEFT C(20), RIGHT C(10), JUNK C(10);
Q(LEFT,RIGHT);
      .
      .
      .
PROC Q(S,T);
ITEM S C(20), T C(20);
S=T;
```

Figure 7-6. Character Strings as Parameters

LABEL NAMES AS PARAMETERS

A label name can be used as an actual parameter. A formal parameter declaration for the label can, but need not, appear in the procedure declaration. It makes debugging easier and is generally good programming practice to declare it, however. The parameter in the transfer vector is assumed to be the address of a label.

PROCEDURE NAMES AS PARAMETERS

A procedure name can be specified as a formal parameter. Within the procedure, the formal parameter declaration is not the same as a procedure declaration elsewhere. The parameter in the transfer vector is assumed to be the address of the entry to the procedure. SYMPL calls the procedure by simulating a return jump.

A formal parameter that is a procedure name must be declared with:

```
FPRC name, name, . . . ;
```

name Identifier of a procedure.

The formal declaration of a procedure name does not include any parameters to that procedure. Such parameters must be established for use in the procedure. Assuming procedures P and S as shown in figure 7-7, a call P(17,S) results in a call to procedure S with 17 as a parameter. The programmer writing procedure P is responsible for knowing that procedure S requires an integer parameter X.

```
PROC P (N, Q);
BEGIN ITEM N;
    FPRC Q;
Q(N);
END # P #

PROC S (X);
BEGIN ITEM X;
      .
      .
      .
END # S #
```

Figure 7-7. Procedure Name as Parameter

A procedure name in a parameter list should be programmed carefully. Since the called procedure must supply parameters and SYMPL checks neither the number or type of

parameters, any execution-time errors are difficult to debug. In the example in figure 7-7, calls to procedure P must supply only the names of the procedures, all of which require exactly the same type of parameters.

ARRAY NAMES AS PARAMETERS

Any array or based array can be specified as a formal parameter. Within the procedure, the formal parameter declaration syntax is the same as array declaration outside of the procedure, including the descriptions of items in the array.

When the formal parameter is specified with a BASED ARRAY declaration, the actual parameter must be a pointer or LOC function, or an expression whose value is a pointer. Access of a formal based array is inefficient and should be avoided. Such access is justified only when the intent of the procedure is to move the based array.

When the formal parameter is specified with an ARRAY declaration, the actual parameter must be an array or based array. An array name can be subscripted; this has the effect of imposing the first element of the formal array onto the designated element of the actual array.

The first word address of an array is passed to a procedure without any accompanying information about array bounds, and SYMPL performs no subscript checking. Consequently, the array bounds are not required in the formal array declaration. The programmer writing the procedure is responsible for bounds and subscript checking.

If the size and structure are not the same for the formal and actual arrays, the wrong elements are accessed. The programmer is responsible for defining the correct field positions in the formal array, and for extracting or storing the desired fields in the actual array.

For single-dimension single-word arrays, bounds can be omitted in the formal declaration since parameters passed to a procedure can control array size. In the example in figure 7-8a, calls to procedure Q set both array A and array B to zero. For multidimension arrays or multiword array items, the formal declaration must be correct to ensure proper results. In figure 7-8b, the first call to procedure P sets array A to zero. The second call, however, erroneously sets more than the 37 items of array B.

EFFICIENCY IN PARAMETER LISTS

The calling sequence for a procedure with parameters is lengthy. Several techniques can be used on source programs to reduce the size of the generated code or to reduce the time required for execution. Three such techniques are: use of call-by-values for scalars or array items, reuse of a single parameter list, and the DEF capability.

a.
```
ARRAY A[0:64]; ITEM AA;
ARRAY B[27:63]; ITEM BB;
Q(A,64);
Q(B,63-27);
      .
      .
      .
PROC Q (X, (N));
BEGIN
ITEM N, I;
ARRAY X; ITEM XX;
FOR I=0 STEP 1 UNTIL N DO
    XX[I]=0;
END # Q #
```

b.
```
ARRAY A[64]; ITEM AA;
ARRAY B[27:63]; ITEM BB;
P(A, 64);
P(B, 63-27);
      .
      .
      .
PROC P (T);
BEGIN
ARRAY T; ITEM TT;
ITEM I;
FOR I=0 STEP 1 UNTIL 64 DO
    TT[I]=0;
END
```

Figure 7-8. Array Names as Parameters

CALL-BY-VALUE PARAMETERS

SYMPL calls subprograms through a return jump instruction. Actual parameters are passed to the subprogram through a transfer vector list.

The address of a parameter list is passed in register A1. If the F parameter appears on the SYMPL compiler call, the last word in each list contains all zeros as required by the FORTRAN Extended calling sequence.

The transfer vector list contains local copies of all parameters used. The two types of parameters are:

A scalar or array item parameter enclosed in parentheses in the formal parameter list indicates that the parameter is to be called by value rather than by address. The transfer vector points to a temporary storage word containing the value. The corresponding actual parameter is protected by SYMPL.

All other parameters appear in the transfer vector lists as addresses of memory words containing their values.

Call-by-address parameters require two memory references to access the parameter. This indirect addressing is less efficient than the direct addressing possible for call-by-value parameters.

For program efficiency, call-by-value should be specified for scalars or array items in a formal parameter list as often as possible. Call-by-address should be used only when the parameter is modified within the procedure and the new value of the parameter is to be returned to the calling subprogram.

REUSING A PARAMETER LIST

The SYMPL compiler uses the same transfer vector as many times as possible. Consequently, the size of generated code can be reduced by rewriting some calls to reference global identifiers. Consider the following:

A declaration for procedure P is identical to that for procedure Q:

```
PROC Q (R, S, T, U, (V));
```

If the calls are P(A, B, C, D, F+1) and Q(A, B, C, D, E), the same transfer vector cannot be used. These two calls do allow the same transfer vector:

```
H=1; P(A, B, C, D, H);

H=F+1; Q(A, B, C, D, H);
```

Use of global identifiers, external identifiers, and common variables must be considered in relation to other modular programming needs.

The IF statement allows alternative statements to execute, depending on whether a Boolean expression is TRUE or FALSE. The FOR statement simplifies coding of repetitive operations.

IF STATEMENT

The IF statement has three clauses:

The IF clause specifies the Boolean condition to be tested.

The THEN clause specifies the statement to execute when the result of the IF clause evaluation is TRUE.

The ELSE clause specifies the statement to execute when the result of the IF clause evaluation is FALSE. This clause is optional; if omitted, the statement following the THEN clause executes when the result is FALSE.

The IF statement syntax is:

IF Boolean expression THEN statement ELSE statement

Boolean expression	Boolean expression specifying the condition to be tested.
statement	Any elementary statement or compound statement. All statements must be terminated with semicolons just as if they were not associated with IF.

Since the ELSE portion of the IF statement is optional, the simplest form of the IF statement is:

IF Boolean expression THEN statement;

Both of these are valid IF statements:

IF A EQ 0 THEN T[I]=0; ELSE T[I]=2;

IF A EQ 0 THEN T[I]=1;

ELSE distinguishes between statements that are always executed and those that execute only when a condition is false.

The logic of the IF statement is shown in figure 8-1. The THEN statement executes only when the Boolean expression is TRUE; the ELSE statement executes only when the Boolean expression is FALSE.

The differences in execution between the following two statements

IF A EQ B THEN C=D; E=F; G=H;

and

IF A EQ B THEN C=D; ELSE E=F; G=H;

appears in the logic diagrams shown in figure 8-2. The second diagram illustrates the execution of E=F; only when A EQ B; is false.

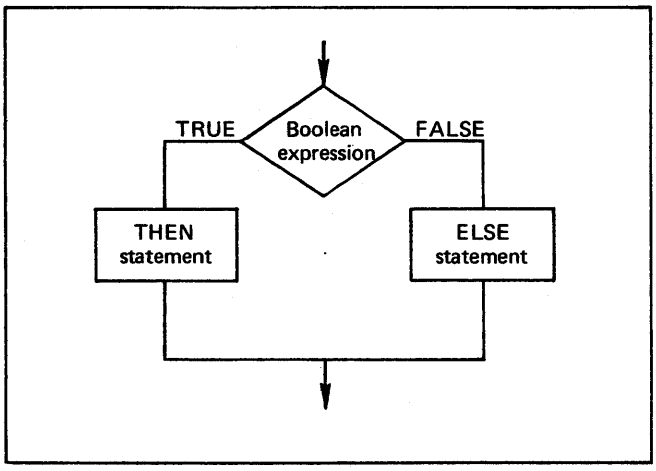


Figure 8-1. IF Statement Logic

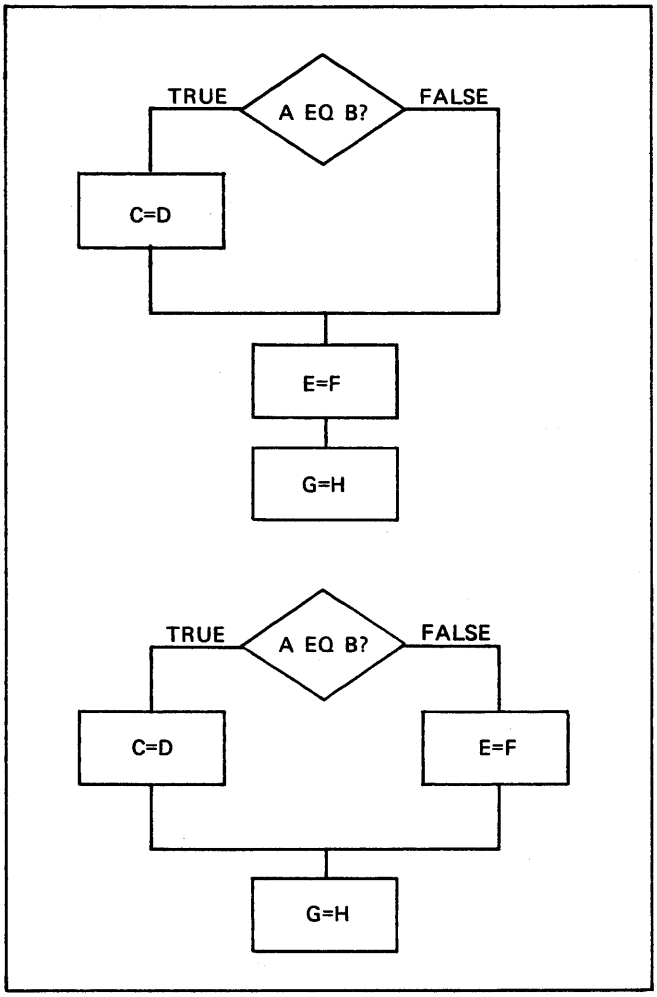


Figure 8-2. ELSE Statement Logic

All the statements in an IF construct are subject to the same rules, including punctuation, as other statements in SYMPL.

The statement can be an elementary statement such as:

```
BIRD="TROCAN";
```

The statement can be a compound statement, as shown in figure 8-3a. The statement can be another IF statement, FOR statement, STOP statement, and so forth, as shown in figure 8-3b.

```
a. BEGIN
   BIRD="TROCAN";
   TREE=24;
   END

b. IF A EQ 0 THEN
       IF B EQ 0 THEN
           C=1;
       ELSE
           C=2;
```

Figure 8-3. IF Statement Example A

A common programming practice is to write every statement following THEN and ELSE as a compound statement. In this instance the BEGIN and END visually delimit nested statements, as shown in figure 8-4.

```
IF A EQ 0 THEN
    BEGIN
    IF B EQ 0 THEN
        BEGIN
        C=1;
        END
    ELSE
        BEGIN
        C=2;
        END
    END
```

Figure 8-4. IF Statement Example B

Punctuation within an IF statement follows the rule that each elementary statement must be terminated by a semicolon. Each statement in the IF construct has a following semicolon. No semicolons are associated with BEGIN and END.

NESTED IF STATEMENTS

When IF statements are nested, the ELSE portion of an IF statement is always associated with the innermost nested IF. A nested IF statement and its corresponding logic flow are shown in figure 8-5.

It is a better practice, however, to write nested IF statements as compound statements to avoid confusion on this point. It makes the code more obvious, and, in terms of execution time and space, is no more costly. The statement in figure 8-5 should be written as shown in figure 8-6.

Another example of a nested IF statement, in which C=4 only if neither A nor B is 0, is shown in figure 8-7.

```
IF A EQ 0
    THEN
    IF B EQ 0
        THEN
        C=1;
        ELSE
        D=1;
```

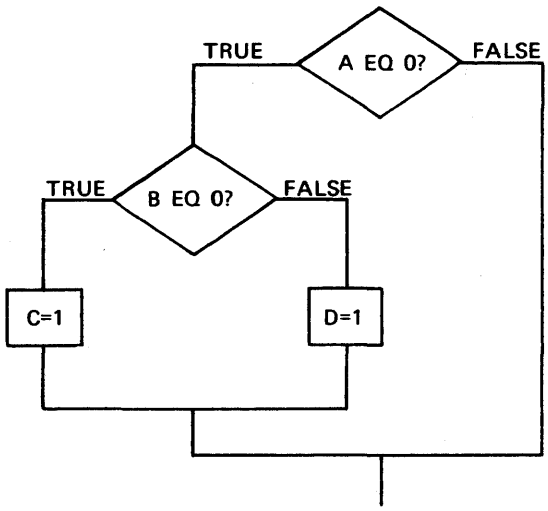


Figure 8-5. Nested IF Statement Example A

```
IF A EQ 0 THEN
    BEGIN
    IF B EQ 0
        THEN C=1;
        ELSE D=1;
    END
```

Figure 8-6. Nested IF Statement Example B

```
IF A EQ 0 THEN IF B EQ 1 THEN C=1;
                         ELSE C=2;
               ELSE IF B EQ 0 THEN C=3;
                         ELSE C=4;
```

Figure 8-7. Nested IF Statement Example C

BOOLEAN EXPRESSIONS IN IF STATEMENTS

Any Boolean expression can be used in an IF statement. Evaluation of the expression terminates as soon as any part of the expression determines the results. The example in figure 8-8a is evaluated as if it were written as shown in figure 8-8b.

This feature avoids wasteful tests and can result in valuable protection in a program. For example, in the following the procedure SQROOT is not called when X is negative:

```
IF X GQ 0 AND SQROOT(X) EQ Y
    THEN ...
```

```
a. IF I GR 0 AND T[I] EQ 0
       THEN X=0;
       ELSE X=1;

b. IF I GR 0
       THEN IF T[I] EQ 0
            THEN X=0;
            ELSE X=1;
       ELSE X=1;
```

Figure 8-8. Boolean Expression in an IF Statement

Evaluation of two Boolean expressions can be forced by an expression of the proper form. For example:

IF A EQ B AND A EQ C THEN . . .

can be written in a faster executing form:

IF (A-B LOR A-C) EQ 0 THEN . . .

However, clarity and maintainability should be considered when code is written for faster execution time.

FOR STATEMENT

The FOR statement should be used any time a statement is to execute at least 3 or 4 times, or any time the conditions for execution might not exist.

The FOR statement has three clauses:

The FOR clause specifies the conditions under which the DO clause is to be executed. Those conditions might result in zero executions.

The WHILE clause or the UNTIL clause specifies the conditions that terminate the DO clause executions. The WHILE clause offers execution advantages in certain cases.

The DO clause specifies the operations to be repeated. In most instances, the DO clause includes a compound statement.

An example of a FOR statement that sets each element of array T to 0 is shown in figure 8-9.

```
DEF SIZE #1024#;
ARRAY [SIZE]; ITEM T;
ITEM I;
FOR I=0 STEP 1 UNTIL SIZE DO T[I]=0;
```

Figure 8-9. FOR Statement Example

The FOR statement is an extension of the DO statement of FORTRAN. It differs from DO in several respects, however. In SYMPL:

The induction variable (loop counter) must be declared as a scalar before it can be used.

The step value can be negative.

A loop is not necessarily executed once.

The TEST statement can be included in the loop to cause remaining computations inside the loop to be bypassed.

A CONTROL statement can affect the optimization the compiler performs with the statement.

SYMPL version 1.2 introduces program control over the code generated for FOR loops. Through a CONTROL FASTLOOP or CONTROL SLOWLOOP compiler-directing statement, a SYMPL 1.2 program can specify the implementation of the loop within each individual FOR statement. Execution advantages can be gained by specifying FASTLOOP; on the other hand, this specification puts restrictions on the format and use of the FOR statement.

SYMPL versions prior to 1.2 always produced slow loops that could not be optimized since the compiler could not ascertain the permanence of all statement characteristics. The default condition for version 1.2 is SLOWLOOP.

When the programmer knows that a loop has certain characteristics, however, CONTROL FASTLOOP should be specified to obtain optimization. The following characteristics are required for optimization:

The induction variable is type integer or type unsigned integer with an absolute value that can be expressed in 17 bits.

The variables in the arithmetic expression of the STEP clause must not be modified.

The variables in the arithmetic expression of the UNTIL clause, if present, must not be modified inside the controlled statement.

The controlled statement of the loop is executed at least once.

The variables in a WHILE clause can be changed in the loop; however, the current value is always used.

For both fast loops and slow loops, the programmer can affect code efficiency by properly planning the loop.

Faster execution for slow loops can be achieved by moving arithmetic expressions outside the loop. In the example in figure 8-10a, TAB[J]-1 is evaluated once, but N+2 and TAB[J] are evaluated every repetition. To avoid evaluation of arithmetic expressions within the loop, the FOR statement in figure 8-10a could be written as shown in figure 8-10b. Further, any call to a procedure or function within a loop inhibits optimization.

```
a. FOR I=TAB[J]-1
       STEP N+2 UNTIL TAB[J]-1 DO
       TAB[I]=0;

b. S=N+2;
   L=TAB[J]-1;
   FOR I=TAB[J]-1
   STEP S UNTIL L DO
   T[I]=0;
```

Figure 8-10. Evaluation of Arithmetic Expression in a FOR Statement

FOR SYNTAX

The general format of the FOR statement is:

FOR induction variable = loop control DO statement;

induction variable — Identifier of data type I, U, S, or R to be used as the loop counter. Data type R is not often used. The type of this induction variable establishes the mode for evaluation of arithmetic expressions in the FOR statement. When the loop terminates, the current value of the induction variable is available only if a jump exits from the loop. If the loop terminates normally, the induction variable is not defined.

loop control — Condition under which the loop is to be executed. It can take several forms as noted below.

statement — Statement to be executed as long as the loop control condition exists. This statement, which is called the controlled statement, can be any elementary or compound statement, including an IF statement or a FOR statement. Good programming practice is to write the controlled statement as a compound statement at all times.

LOOP CONTROL

For slow loops, evaluation of the test condition occurs at the beginning of each loop before the controlled statement is executed. Consequently, the controlled statement can be bypassed. In the following example T[I]=0 is never executed:

```
L=3;
FOR I=4 STEP 1 UNTIL L DO
    T[I]=0;
```

Both the test for loop terminating conditions (WHILE Boolean expression or UNTIL arithmetic expression) and the increment to the induction variable take place within the loop.

The loop control has these five forms (the fourth and fifth forms produce an infinite loop; the programmer is responsible for coding an exit jump):

1. Initial WHILE Boolean expression
2. Initial STEP arithmetic expression WHILE Boolean expression
3. Initial STEP arithmetic expression UNTIL arithmetic expression
4. Initial STEP arithmetic expression
5. Initial

initial — Arithmetic expression giving the initial value of the induction variable. The expression is evaluated once at the start of the FOR statement.

Boolean expression — Boolean expression specifying the condition under which looping is to continue. As long as the expression is TRUE, looping continues; when the expression is FALSE, looping does not take place.

arithmetic expression — Arithmetic expression indicating:

STEP Clause — Increment to the induction variable to be added each loop. This constant or variable can have a positive or negative value.

UNTIL Clause — Value after which looping terminates.

The expression can have a negative, as well as a positive, value.

The logic of a statement with a WHILE clause and STEP clause with a slow loop is shown in figure 8-11. For an UNTIL clause with a positive step with a slow loop, the logic is as shown in figure 8-12. Figure 8-13 shows the logic of an UNTIL clause with a fast loop, when a STEP expression can be positive or negative.

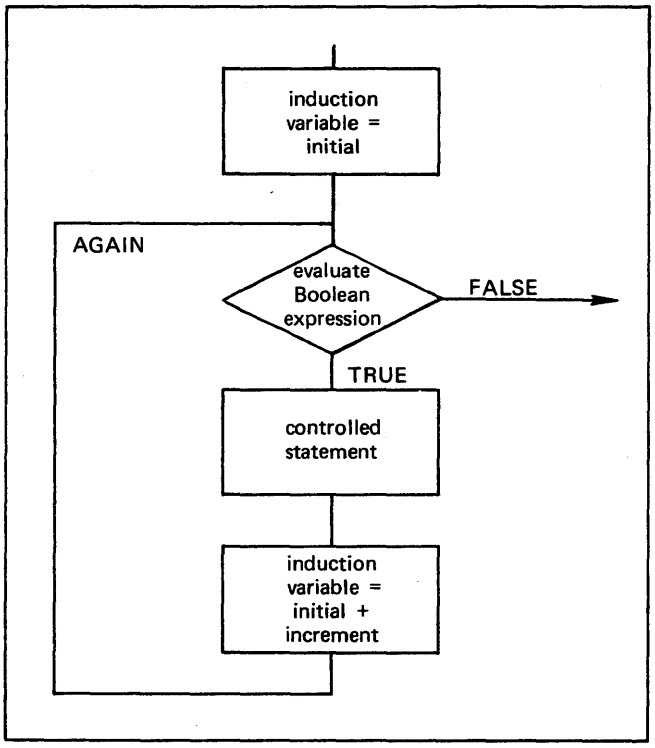


Figure 8-11. Slow Loop Logic Example A

WHILE Clause

The WHILE clause of the FOR statement combines the capabilities of an IF statement with the looping capabilities of FOR. For example, the sequence shown in figure 8-14 assigns SOL the minimum value of I, if any, when T[I]=0.

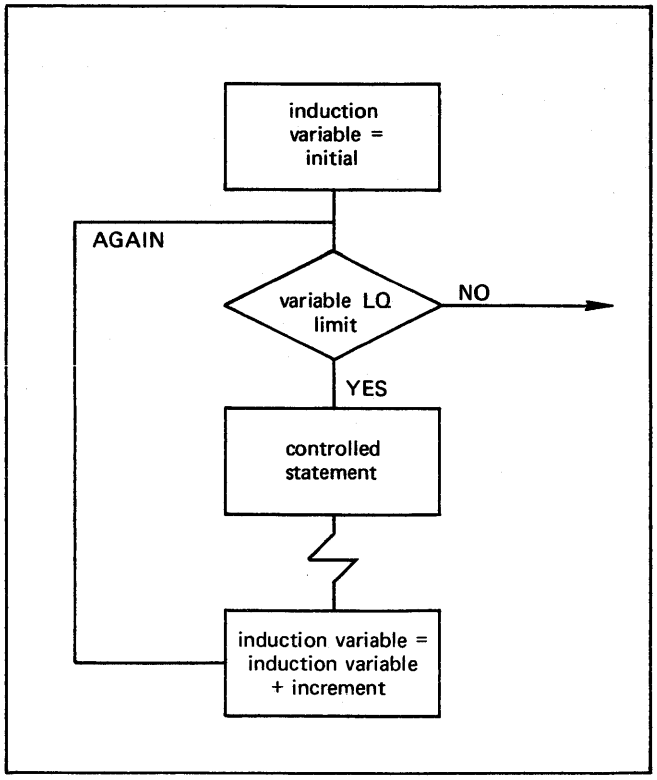


Figure 8-12. Slow Loop Logic Example B

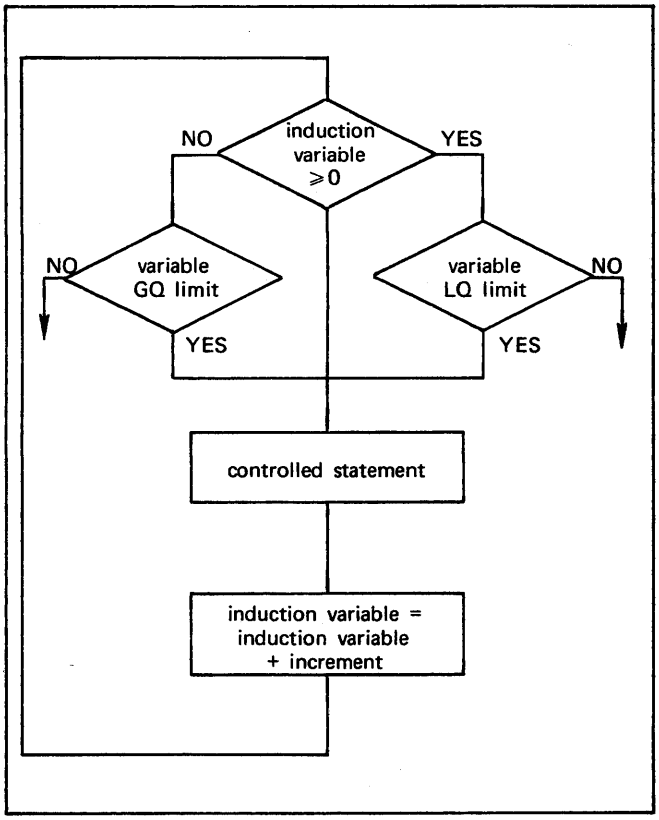


Figure 8-13. Fast Loop Logic Example

```
SOL=0;
FOR I=0 STEP 1 UNTIL 100 DO
    IF T[I] EQ 0
    THEN
        GOTO FOUND;
    .
    .
    .
FOUND: SOL=I;
```

Figure 8-14. WHILE Clause Example A

Using the WHILE clause, a FOR statement can be written to accomplish the same function if it is certain that T[I]=0 exists to stop the loop:

```
SOL=0;
FOR I=0 STEP 1 WHILE T[I] NQ 0 DO
    SOL=I;
```

Empty compound statements are often useful in FOR statements with WHILE clauses. For example, the statement shown in figure 8-15 exits from the loop with MIN having the minimum value such that T[MIN] exceeds 0. This technique is valid only with a slow loop. With a fast loop the value of the induction variable is undefined on a normal exit.

```
FOR MIN=0 STEP 1
WHILE T[MIN] LQ 0 DO
    BEGIN
    END
```

Figure 8-15. WHILE Clause Example B

No form of the FOR statement exists in which the reserved words FOR and the initial value of the induction variable can be omitted. That is, a WHILE B DO statement is not valid. The same results can be achieved, nevertheless, through use of the DEF statement to generate a valid FOR statement, as shown in figure 8-16.

```
ITEM DUMMY;
DEF ASLONGAS #FOR DUMMY=DUMMY WHILE#;
#SET NEXT TO FIRST ELEMENT OF LIST#
ASLONGAS NEXT NQ 0 DO
    BEGIN
    .
    .
    .
    END
```

Figure 8-16. WHILE Clause Example C

Controlled Statement

The controlled statement can be any valid statement. Examples of common types of controlled statements are given below; they assume all variables have been defined previously.

1. The statements necessary to initialize three arrays are shown in figure 8-17.

2. The IF statement as a controlled statement is illustrated in figure 8-18.

3. The FOR statement as a controlled statement which sets the lower triangle of MATRIX to 0 is shown in figure 8-19.

4. A compound statement nesting within a controlled statement is shown in figure 8-20.

```
FOR I=1 STEP 1 UNTIL N DO
    BEGIN
    T[I]=0;
    U[I]=0;
    V[I]=I;
    END
```

Figure 8-17. Controlled Statement Example A

```
FOR I=M STEP 1 UNTIL N DO
    IF T[I] EQ 0
    THEN
    GOTO L;
    ELSE
        BEGIN
        K=K + 1;
        U[K]=U[K] + 1;
        END
```

Figure 8-18. Controlled Statement Example B

```
ARRAY [1:10, 1:10]; ITEM MATRIX;
FOR I=1 STEP 1 UNTIL 10 DO
    FOR J=1 STEP 1 UNTIL I DO
        MATRIX[I,J]=0;
```

Figure 8-19. Controlled Statement Example C

```
FOR I=1 STEP 1 UNTIL N DO
    BEGIN
    TAM[I]=0;
    FOR J=1 STEP 1 UNTIL N DO
        BEGIN
        MAT[I,J]=0;
        TAB[I,J]=TAB[I,J] + 10 * I + J;
        END
    END
```

Figure 8-20. Controlled Statement Example D

A jump out of the controlled statement is valid. Under such circumstances, the current value of the induction variable is preserved and can be used outside the statement.

A jump into a controlled statement from outside the controlling FOR statement is possible, although such an action generally has no meaning and produces errors. Although the induction variable can be modified within the controlled statement on slow loops, good programming practice avoids such code.

TEST STATEMENT OF FOR

The TEST statement has meaning only within the FOR controlled statement. TEST, which allows the remaining part of a loop to be bypassed, is equivalent to a FORTRAN statement that jumps to a CONTINUE statement in a DO loop.

The use of TEST, in which the statement V[I]=0 is bypassed for values of I such that U[I]=VAL, is illustrated in figure 8-21. Without TEST, the sequence shown in figure 8-21 appears as shown in figure 8-22.

When loops are nested, the induction variable name can be added to the TEST statement to specify which loop is to be bypassed, as illustrated in figure 8-23. The logic of the code in figure 8-23 is as if it were written as shown in figure 8-24.

```
FOR I=0 STEP 1 UNTIL N DO
    BEGIN
    T[I]=0;
    IF U[I] EQ VAL
        THEN
        TEST;
    V[I]=0;
    END
```

Figure 8-21. TEST Statement Example A

```
    I=0;
AGAIN:
    IF I LQ N
        THEN
        BEGIN
        T[I]=0;
        IF U[I] EQ VAL
        THEN
        GOTO NEXT;
        VAL[I]=0;
NEXT:
        I=I + 1;
        GOTO AGAIN;
        END
```

Figure 8-22. TEST Statement Example B

```
FOR I=0 STEP 1 UNTIL N DO
    FOR J=0 STEP 1 UNTIL M DO
        BEGIN
        A[I,J]=A[I,J] + 1;
        IF A[I,J] EQ VAL THEN TEST I;
        IF A[I,J] EQ LAV THEN TEST J;
        B[I,J]=0;
        END
```

Figure 8-23. TEST Statement Example C

```
AGAINI:  IF I LQ N THEN
             BEGIN
             J=0;
AGAINJ:  IF J LQ M THEN
             BEGIN
             A I,J=A I,J + 1;
             IF A I,J EQ VAL THEN GOTO NEXTI;
             IF A I,J EQ LAV THEN GOTO NEXTJ;
             B I,J=0;
NEXTJ:       J=J+1; GOTO AGAINJ;
             END
NEXTI:       I=I+1; GOTO AGAINI;
             END
```

Figure 8-24. Logic of TEST Statement

COMPILATION CONTROL 9

SYMPL compilation is controlled by:

DEF statements in the program that are similar to COMPASS macros, as described in section 5.

CONTROL compiler-directing statements in the program.

$BEGIN and $END debugging code delimiters.

SYMPL compiler call itself.

The CONTROL statement is a compiler-directing statement rather than an executable statement in a program. The words used in the CONTROL statement are not reserved words: ITEM NOLIST, for example, is legal. Also, these words can be expanded by DEF.

Several types of actions are influenced by CONTROL, including:

Source listing control.

Compilation options affecting packed switches, preset of common, and FORTRAN compatibility.

Characterization of variables and arrays for optimization purposes.

Conditional assembly.

Table 9-1 shows all control-words of the CONTROL statement and the range of compiler action in regard to each statement.

TABLE 9-1. CONTROL-WORDS OF CONTROL

Control-Word	Function	Extent of Effect
DISJOINT variable	Characterize variable as having single name.	Entire module
EJECT	Skip to new page of source listing output.	Single compiler action
ENDIF	End conditional assembly begun by IFxx.	Immediate compiler action
FASTLOOP	Generate FOR loop similar to FORTRAN DO loop.	Until a subsequent FASTLOOP, SLOWLOOP, or TERM
FI	Same as ENDIF.	Same as ENDIF
FTNCALL	Turn on compiler call F parameter.	Entire module
IFxx condition	Compile code if condition true.	UNTIL balanced ENDIF or FI
INERT array	Characterize array as not having overlapping subscript references.	Entire module
LEVEL n block	Specify memory residence of common block or based array.	Entire module
LIST	Resume source listing.	Until subsequent NOLIST or TERM
NOLIST	Suspend source listing unless H parameter on compiler call or OBJLIST appears.	Until subsequent LIST or TERM
OBJLIST	List object code, overlapping compiler call list parameter and LIST or NOLIST.	Entire module
OVERLAP variable	Characterize variable as having more than one name.	Entire module
PACK	Pack switch code.	Entire module
PRESET	Preset items in common.	Entire module
REACTIVE array	Characterize array as possibly having overlapping subscript references.	Entire module
SLOWLOOP	Generate FOR loop without FORTRAN similarities.	Until a subsequent FASTLOOP, SLOWLOOP, or TERM
TRACEBACK	Generate traceback information.	Entire module
WEAK name	Characterize name as being a weak external.	Entire module

CONDITIONAL COMPILATION

The IFxx form of the CONTROL statement allows conditional compilation that resembles COMPASS conditional assembly. SYMPL offers fewer capabilities than COMPASS, with no statements equivalent to COMPASS pseudo-instructions ELSE and IF DEF.

For instance, to compile source statements only when DBUG=0 in COMPASS and SYMPL, the statements shown in figure 9-1 can be used. In each case, the code that produces an error message and aborts the program is not assembled when DBUG=0. The COMPASS code that conditionally assembles the range identified by name B is in figure 9-1a. The same function in SYMPL is performed as shown in figure 9-1b.

```
a.      DBUG EQU 1
            .
            .
            .
        B   IFNE DBUG,0
            SA1  MSGVECT
            RJ   ERROR
            JP   ABORT
        B   ENDIF

b.      DEF   DBUG #1#;
            .
            .
            .
            CONTROL IFNQ DBUG,0;
            ERROR(MESSAGE);
            GOTO ABORT;
            CONTROL ENDIF;
```

Figure 9-1. CONTROL Statement Example A

In both languages, the conditional source statements are bracketed between a statement defining the conditions and a statement ending conditional assembly. In SYMPL, the ending statement can be either:

CONTROL ENDIF; or CONTROL FI;

However, this statement must not be generated by a DEF. When the IF condition is false, DEF statements are not expanded.

The format of a conditional assembly statement is:

CONTROL IFxx constant1, constant2;

xx — Condition that compiler is to test constants for in a constant1 xx constant2 situation:

EQ	Equal
LS	Less than
LQ	Less than or equal to
GR	Greater than
GQ	Greater than or equal to
NQ	Not equal

constant1, constant2 — Constants or status functions to be tested. Generally, at least one constant is defined through DEF.

Both constants must be the same type since SYMPL does not convert types in this context. Data type B and C should be compared only with IFEQ and IFNQ. Blanks are significant in character strings, whether the blanks are within the string or at the end of the string.

If only one constant appears, it is assumed to be constant1, and constant2 is assumed to have a value of 0.

When the condition is false, assembly continues with the next statement after the balancing CONTROL ENDIF of CONTROL FI. The source listing produced shows a minus sign in the left margin.

An example in which code is generated to call procedure S when FAST=0 is shown in figure 9-2.

```
DEF FAST #0#;
.
.
.
CONTROL IFEQ FAST;
S;
CONTROL FI;
```

Figure 9-2. CONTROL Statement Example B

A capability similar to ELSE of COMPASS can be simulated by the negation of the direct IF control statement. In the example in figure 9-3, MODEL is defined through DEF (as in DEF MODEL #76# or DEF MODEL #74#). Depending on the model, a one-bit is tested for 0 or 1.

```
CONTROL IFEQ MODEL, 76;
IF B<MFLAG> WORD [OPTS] EQ 0
    THEN RETURN;
CONTROL ENDIF;
CONTROL IFNQ MODEL, 76;
IF B<MFLAG> WORD [OPTS] EQ 1
    THEN RETURN;
CONTROL FI;
```

Figure 9-3. CONTROL Statement Example C

Similarly, a logical product (AND) of conditions can be satisfied by nested CONTROL statements. In the example in figure 9-4, a call to LOAD (TBL, XDEFNAME, FALSE) is generated when the model is not 76 and the system is neither ATS nor KRONOS. Notice that DEF is used within the conditional code to redefine SKIP.

OPTIMIZATION CONTROL

The SYMPL version 1.2 compiler introduced four CONTROL statement control-words that can be used to influence optimization performed by the compiler. None of these statements (OVERLAP, DISJOINT, INERT, REACTIVE) is required. In their absence, the compiler proceeds with its normal optimization. Because the consequences of some optimizations are unpredictable, default optimization is limited.

When the programmer informs the compiler that variables and array subscripts have been limited to uses with known consequences, the additional optimization can occur. Programs with such limits are called behaved, as opposed to unbehaved programs.

```
STATUS SYS ATS, INTCOM, KRONOS, S34, S2;
DEF SYSTEM . . . ;
DEF MODEL . . . ;
.
.
.
CONTROL IFNE MODEL, 76;
    DEF SKIP #1#;
    CONTROL IFEQ SYSTEM, SYS"ATS";
        LOAD (TAB, XDEFNAME);
        DEF SKIP #0#;
    CONTROL ENDIF;
    CONTROL IFEQ SYSTEM, SYS"KRONOS";
        LOAD (TAB, XDEFNAME, TRUE);
        DEF SKIP #0#;
    CONTROL ENDIF;
    CONTROL IFNE SKIP;
        LOAD (TAB, XDEFNAME, FALSE)
    CONTROL ENDIF;
CONTROL ENDIF;
```

Figure 9-4. CONTROL Statement Example D

The SYMPL Reference Manual contains details of the compiler optimization and the use of the optimization control-words. Future versions of the compiler might require these statements.

With or without the optimization CONTROL control-words, the SYMPL compiler performs optimization that moves code as it sees fit. A SYMPL programmer should not assume locations of any executable code.

To allow more, rather than less, optimization, a programmer should consider:

Initialization of a program in one procedure and the body of a program in another. (SYMPL does not move code from one procedure to another.)

Limiting of array subscripts to the bounds of the array, so that A[n] and B[m] are not the same word.

$BEGIN/$END DEBUGGING COMPILATION

Statements in a source program that are delimited by $BEGIN and $END are compiled only when the E parameter is specified on the SYMPL compiler call. Without the E parameter, such statements are shown in the source listing with a minus sign in the left margin, but they are not compiled. The $END statement must not be generated by a DEF. DEF is not expanded within $BEGIN and $END without the E parameter.

An example of this feature used to affect error output is shown in figure 9-5. CURSTAT is not allocated any memory space unless the E parameter is selected. The check of BYTETYP NQ S"INT" always compiles; in debug mode it produces a message, and in normal mode it does nothing.

SYMPL COMPILER CALL

The SYMPL compiler calls follow the conventions of other language processors, with I=INPUT, L=OUTPUT, and B=LGO parameter defaults. The compiler call using all defaults is:

SYMPL.

```
PROC PASSN;
BEGIN
    $BEGIN
    ITEM CURSTAT;
    $END

    PROC MISTAKE(CODE, AUX1, AUX2);
    BEGIN
    ITEM CODE, AUX1, AUX2;
        $BEGIN
        ERPRINT(CODE, CURSTAT, AUX1, AUX2);
        RETURN;
        $END
    END #PROC MISTAKE#
.
.
.
IF BYTETYP NQ S"INT"
THEN MISTAKE(ERR"NOTINT",IN[0], INX);
.
.
.
INPARMX=INPARMX+1;
    $BEGIN
    CURSTAT=CURSTAT+1;
    IF INPARMX GR INTYPE"MAX"
    THEN ERROR(ERR"INMAX");
    IF DEBUG0 THEN TRNACINT(INPARMX);
    $END
.
.
.
END #PROC PASSN#
```

Figure 9-5. Use of $BEGIN and $END

Other compiler call parameters are summarized in table 9-2. The SYMPL Reference Manual describes all parameters in detail.

Listings are controlled by any combination of LXOR=lfn:

X	Storage map and common block listing
O	Object code, lfn/line/line lists only code for source lines indicated by number
R	Cross reference map and common block listing

The time required to compile a program depends more on the length of the source code than on the number of declarations. On a CYBER 70 Model 73 system, about 2000 lines can be compiled per minute when full compilation is selected.

The total field length required for a given compilation depends on the length of the symbol table which, in turn, is dependent on the number of declarations rather than length of source code or statements. For each entry in the table, five words are required.

Field length requirements are, at minimum:

51K octal under NOS 1 and NOS/BE 1

41K octal under SCOPE 2

The SYMPL compiler is written, for the most part, in SYMPL.

Generated code might reference the FORTRAN library and SYMLIB (NOS 1 and NOS/BE 1) or SYMIO (SCOPE 2) library. The FORTRAN library is expected to contain routines XTOI and ITOJ for exponentiation and routines for print input/-output. The SYMLIB or SYMIO library is expected to contain the SYMPL execution-time routines SYMSM$, SYMSC$, and SYMSG$ for the more complex bit and character processing routine, SYMBSW$ for switch checking, and the SYMPL interface routines to the print facilities.

TABLE 9-2. COMPILER CALL PARAMETERS

Parameter	Significance
A	Abort after error
C	Check switch references for range
D	Pack switches two per word
E	Compile debugging statements within $BEGIN and $END
F	Generate procedure call parameter lists compatible with FORTRAN Extended
H	List all source statements despite any CONTROL NOLIST statement
I	Designate input file to be other than INPUT
K	Generate points-not-tested interface code
N	List unreferenced items on cross reference map
P	Initialize (preset) items in labeled common
S=0	Suppress LDSET table generation
S=lib/lib	Generate LDSET table with entries for named libraries. Default is S=SYMLIB/-FORTRAN for NOS 1 and NOS/BE 1; S=SYMIO/FORTRAN for SCOPE 2.
T	Suppress code generation
W	Single statement scheduling for closer correspondence between source statement order and object code order
Y	Suppress diagnostic 136, SEMI ENDS COMMENT

DEFINITIONS 1

1.1 SYNTAX NOTATION

The syntax notation used in the formal presentation of the PASCAL language elements adheres to the following rules:

1. Keywords appear in capital letters and must be specified exactly as shown. In the examples presented, the keywords have been underlined.
2. Variables and procedure names appear as single or a hyphenated set of lowercase words that suggest the meaning attributed to them.
3. All non-alphanumeric characters, except the hyphen, must appear as presented.
4. Optional items appear enclosed in slashes, e.g., /optional-item\.
5. One or more repetitions of a single syntactical item is indicated by following it with a succession of three periods (...).

1.2 CONSTANTS

A constant is a literal representation of a fixed value that is associated with some data type. A PASCAL constant may be specified as:

- A decimal integer number
- A hexadecimal integer number
- An octal integer number
- A character
- A character string

A decimal integer is specified as a string of decimal digit characters not including a decimal point. It may be preceded by a sign (+ or -). The general form of a decimal integer is:

/sign\ decimal-digit /decimal-digit\...

A hexadecimal integer is specified as a string of hexadecimal digit characters preceded by a dollar sign ($). A hexadecimal integer may not have a preceding sign nor may it include a hexadecimal point. The general form of a hexadecimal integer is:

$hex-digit /hex-digit\...

An octal integer is specified as a string of octal digit characters followed by the letter B. An octal integer may not have a preceding sign nor may it include an octal point. The general form of an octal integer is:

octal-digit /octal-digit\ ...B

The range of values that may be specified as a decimal constant, a hexadecimal constant, or an octal constant are:

-32,767 ≤ decimal-integer ≤ 32,767

$0 ≤ hexadecimal-integer ≤$FFFF

0B ≤ octal-integer ≤ 177777B

Some sample integer constant specifications are:

1 0 100 -5273 $7C $3121 77B

A character constant consists of a single graphic character enclosed in single triple marks. A character string constant consists of two or more graphic characters enclosed in single triple marks. The general form of a character constant is:

≡graphic≡

The general form for a character string is:

≡graphic graphic /graphic\...≡

Note that it is necessary to represent the triple mark graphic by a pair of single triple marks.

Some example character constants and character strings appear as:

≡A≡ ≡$≡ ≡≡≡≡ ≡A STRING≡ ≡2B≡

1.3 IDENTIFIERS

An identifier represents constants, type definitions, variables, procedures, and functions to the PASCAL compiler. It is a contiguous sequence of letters and decimal digits that begins with a letter. An identifier is delimited by characters that are neither letters nor decimal digits.

An identifier may be freely chosen (with the exception of reserved keywords) by the programmer wherever the PASCAL language prescribes. It may be any number of characters in length, but its uniqueness must be apparent within the first six characters.

Some sample identifiers are:

BEWARE A123 FORTHRIGHT

2BAR is not an identifier.

SAMEID and SAMEIDENTIFIER are viewed as identical identifiers by the PASCAL compiler.

1.4 RESERVED KEYWORDS

PASCAL uses keywords to direct the compilation process. Each keyword takes the form of an identifier, but is viewed as a distinct, special symbol. The keywords cannot be used as identifiers, so they are considered to be reserved. The following keywords are reserved for PASCAL:

IF	END	THEN	BEGIN	REPEAT	FORWARD	FUNCTION	PROCEDURE
DO	NIL	ELSE	UNTIL	DOWNTO		RELATIVE	
TO	FOR	GOTO	WHILE	RECORD			
OF	DIV	CASE	ARRAY	PACKED			
IN	MOD	WITH	VALUE				
	VAR	TYPE	CONST				
	SET	FILE	LABEL				
	NOT						

SYMPL has no input/output facilities. The SYMPL library does, however, contain procedures that are links to the PRINT routines of FORTRAN Extended.

To use the output features:

A FORTRAN Extended main program must call the SYMPL subprogram. The PROGRAM statement of the main program must specify the file OUTPUT.

The SYMPL program must specify the library procedures in an XREF declaration. Procedures PRINT and ENDL always are required; LIST is optional.

The SYMPL program must call both procedure PRINT and procedure ENDL for each output list to be printed. If variables are to be output, a LIST procedure call is required for each variable. PRINT, LIST, and ENDL form a single output sequence and must appear in that order, although intervening statements can appear.

The library procedures have alternative names PRINT$, LIST$, and ENDL$ for use when PRINT, LIST, or ENDL conflicts with a name used elsewhere in a program. The required externals are specified with an XREF declarative as shown in figure 10-1.

```
XREF BEGIN
     PROC PRINT;
     PROC LIST;
     PROC ENDL;
     END
```

Figure 10-1. Output XREF Declarations

The parameters for the SYMPL procedure calls are based on the FORTRAN statements. A FORTRAN Extended PRINT statement and its associated FORMAT statement have this format:

```
      PRINT label, parameter1, parameter2, . . .
label FORMAT (format specification)
```

The label of the FORMAT statement is not required for SYMPL output. The format specification specifies the format in which the parameters are to be output, including carriage control or Hollerith constant specifications. In SYMPL, this entire format, including its enclosing parentheses, must appear as a character string in a PRINT procedure call. Each FORTRAN parameter specifies a variable or array to be printed. In SYMPL, each item or array to be printed must appear in an individual LIST procedure call.

Any errors in the format specification and LIST arguments are detected during execution by the FORTRAN routines. The FORTRAN Extended Reference Manual explains any error messages that might result.

PRINT PROCEDURES

PRINT specifies the format in which information is to be output. Information appears on the file OUTPUT. Another library procedure, PRINTFL, is available for writing to files other than OUTPUT, as described in the SYMPL Reference Manual for PRINTFL discussion. The procedure call is:

```
PRINT("(specification)");
```

specification — String of characters duplicating the specification of a FORTRAN Extended PRINT statement. The specification can be any legal FORTRAN specification. Parentheses are required to be part of the string.

Examples of PRINT procedure calls are:

Assume a literal is to be printed. Either of the following can be specified:

```
PRINT ("(10H DISASTER)");

PRINT ("(* DISASTER*)");
```

Assume a character string item defined by:

```
ITEM SYNTABFORM C(40)=
#(6H HASH=O6, 11X6, 6HIDENT=2A10, . . .)#;
```

The string can be specified by simply:

```
PRINT(SYNTABFORM);
```

Assume an array item defined by ARRAY [1:9]; ITEM NDIGITS C(2)=[#I 1#,#I2#, . . . ,#I9#];

The entire array is specified by:

```
PRINT (NDIGITS[I]);
```

LIST AND ENDL PROCEDURE CALLS

LIST identifies one expression to be output. The procedure call is:

```
LIST(expression);
```

expression — Any item, subscripted array item, or expression to be output.

LIST must follow a PRINT procedure call or another LIST call. One LIST call must appear for each variable element of the PRINT specification.

The order of execution of the multiple LIST calls must correspond to the format of the preceding PRINT statement, just as the output specifications of a FORTRAN FORMAT statement must correspond to the order of parameters in the FORTRAN PRINT statement.

ENDL is required to end each output list. If no LIST calls appear, ENDL is still required. The procedure call is:

```
ENDL;
```

EXAMPLES

1. FORTRAN Extended statements and SYMPL statements that produce the same result are shown in figure 10-2a and figure 10-2b, respectively.

```
a.      PRINT 10
     10 FORMAT (*1 LIST OF IDENTIFIERS *)
        PRINT 20, LNAME, RNAME, HASH
     20 FORMAT (1H0,2A10,3X,I2)

b.      PRINT ("(*1 LIST OF IDENTIFIERS*)");
        ENDL;
        PRINT ("(1H0,2A10,3X,I2)");
        LIST(LNAME); LIST(RNAME); LIST(HASH);
        ENDL;
```

Figure 10-2. Output in FORTRAN and SYMPL

Output written is: LIST OF IDENTIFIERS lname rname hash where LNAME and RNAME are each 10 alphanumeric characters and HASH is a two-digit integer. In the SYMPL code, each variable is a parameter to a LIST procedure call.

2. The SYMPL code to output FATAL or NON-FATAL, depending on the current value of B, is shown in figure 10-3.

```
PRINT("(LX,AL)");
IF B THEN STR="FATAL";
     ELSE STR="NON-FATAL";
LIST(STR);
ENDL;
```

Figure 10-3. SYMPL Output Example A

3. To repeat the format for each iteration of a loop, the FORTRAN Extended routines perform the implicit DO loop, as shown in figure 10-4.

```
PRINT ("(11X,I10)");
FOR I=STKTOP STEP -1 UNTIL 0 DO
    LIST (STACK[I]);
ENDL;
```

Figure 10-4. SYMPL Output Example B

4. The SYMPL code to list array FLAG is shown in figure 10-5.

```
PRINT("(LX,10L3)");
FOR I=1 STEP 1 UNTIL 10 DO
    LIST(FLAG[I]);
ENDL;
```

Figure 10-5. SYMPL Output Example C

PASCAL LANGUAGE DESCRIPTION 2

The programming language PASCAL is a high-level, algorithmic type language. It is patterned after ALGOL 60 and retains the attractive features of that language. Because PASCAL's grammar is essentially context-free, its syntax is unambiguous and simple to define. In addition, the block-oriented structure of ALGOL 60, which is particularly adaptable to structured programming techniques, is preserved.

The two basic constituents of a PASCAL program are the statement and the declaration/definition. Statements indicate the various actions that are to be carried out by a program, and declarations/definitions describe the meaning attached to the various identifiers used in a program.

PASCAL provides some noteworthy extensions to the capabilities of ALGOL 60. PASCAL supports a broad variety of structured statements, so repetitive or conditional actions may be coded in a concise and natural manner. More significantly, ALGOL 60's deficiency in the area of structured data is remedied by the introduction of a set of data structuring techniques. PASCAL also provides pointer-type variables, along with the ability to allocate storage dynamically and explicitly. These features extend the range of applicability of PASCAL beyond its more traditional ancestor.

2.1 PROCEDURES

The procedure is the primary unit of a program structure in PASCAL. It represents the algorithm intended for execution on the processor. A PASCAL procedure is analogous to a subroutine in that it is called and it may have calling sequence parameters associated with it.

Each PASCAL procedure is represented by a declaration/definition (D/D) segment and an action segment. The D/D segment defines the identifiers used within the procedure and the variable data considered local to the procedure. The action segment defines the logic by which the variables will be affected. The basic form of a procedure declaration appears as

```
PROCEDURE heading information
      D/D segment
BEGIN
      action segment
END
```

A procedure may also contain procedure declarations that are coded in line. This type of procedure, which is internal to another procedure, is called a nested procedure and is considered to be local. The procedure in which the local procedure resides is considered global in relation to the local procedure. Any procedure may contain no, one, or many local procedures. A local procedure may contain other local procedure declarations.

The basic form of a procedure declaration with local procedures appears as

```
PROCEDURE heading information
      D/D segment
      /local procedure declarations\
BEGIN
      action segment
END
```

The terms local and global are relational, depending on the frame of reference. A procedure that contains local procedure declarations may be defined as global; however, it may have been declared local to another procedure in which it was defined.

2.2 PROCEDURE HEADING

The heading of a procedure declaration provides the following:

1. It indicates the start of the procedure declaration.
2. It assigns the identifier by which this procedure may be called via a procedure statement.
3. It defines the form of the calling sequence that must be presented with the identifier when this procedure is called.

The general form of the procedure is:

```
PROCEDURE proc-id /formal parameter section\;
```

where proc-id is the assigned procedure identifier.

The formal parameter section, if present, is enclosed in parentheses. The main constituents of the formal parameter section are the formal parameters. These are identifiers that represent the actual parameters (those presented to this procedure when it is called) to be substituted within the procedure program segment when it is to be executed.

When coding a procedure, the formal parameters represent the forms of the calling sequence, establishing the positional relationship the actual parameters must take when this procedure is called. The formal parameters are used in the body of the program segment where the corresponding actual parameters are to be substituted.

The formal parameters, as they appear in the procedure heading, are grouped into two major categories: variable parameters and value parameters.

All formal parameters that appear in the formal parameter section are grouped by category and listed according to the following convention:

1. Formal parameters of the same category and type may be listed together, separated by commas.

2. A type identifier must be declared following the parameter list by specifying a colon immediately after the last parameter and before the type identifier.

3. The variable category designation is explicitly declared by preceding its parameter list with the keyword VAR. The value category designation is implicityly declared by absence of a keyword.

4. Category groups are separated by semicolons.

2.2.1 VARIABLE PARAMETERS

Variable parameters represent data elements for which the procedure may produce a result. That is, when this procedure is called, the actual parameter represents a location where the called procedure may store data. The actual parameter in this case must always be a variable identifier; it may not be a data constant. The general form for specifying a formal parameter list in the variable parameter is:

```
VAR variable-param /, variable-param\...: type-identifier
```

2.2.2 VALUE PARAMETERS

Value parameters represent data elements whose value may be used by the procedure during processing. The procedure will not affect the contents of the parameter as it appears in the procedure that presented it. The actual parameter is an expression. The general form for specifying a formal parameter list in the value parameter category is:

```
value-param /,value-param\...: type-identifier
```

Value parameters are recognized by the absence of a preface keyword.

A procedure can change the value of a presented value parameter during its processing and still not change its value as it exists in the calling procedure. The presented value parameter is copied into a local variable area upon entry into the called procedure and then only the specified local variable is accessed whenever it appears in the program segment.

2.2.3 PROCEDURE HEADING EXAMPLES

With no formal parameter section:

```
PROCEDURE  BININT;
PROCEDURE  LOOPC;
```

With a formal parameter section:

```
PROCEDURE   GETBNS (VAR BINPTR:BPTR);
PROCEDURE   MULPLY (X, Y: INTEGER; VAR Z: INTEGER);
PROCEDURE   FIXFLT (VAR X: CHAR; VAR Y: INTEGER);
```

2.3 DECLARATION/DEFINITION (D/D) SEGMENT

The D/D segment provides preparatory information to the PASCAL compiler for its use in generating the object text defined in the action segment. Five categories of information may be provided in the D/D segment:

- Label declarations
- Constant definitions
- Type definitions
- Variable declarations
- Value initialization

Each category is presented within the D/D segment headed by the appropriate keyword. The presentation of each category is optional; they must be presented in the order shown above.

2.3.1 LABEL DECLARATIONS

The label declaration specifies all statement labels that are:

1. Defined in the procedure's action part
2. Referenced by a GOTO statement in procedures that are themselves local to the procedure

The general form of a label declaration is

```
LABEL statement-label /, statement-label\...;
```

Examples:

```
LABEL   4;
LABEL   5, 15, 3;
```

2.3.2 CONSTANT DEFINITIONS

A constant definition equates a constant value to an identifier where a constant value may be:

1. A literal constant
2. A previously defined constant identifier
3. A standard constant identifier

A constant definition does not generate data but simply assigns a symbolic identifier to represent the constant value. The constant identifier is thus synonymous with the constant value. An analogous capability is available with most assemblers via the pseudo-operation usually known as EQUATE or EQU.

PASCAL provides a set of standard (predefined) constant identifiers:

- NIL - Represents a pointer value that points to no element at all. For PASCAL, NIL is represented by a zero word.
- TRUE - Represents the TRUE value condition for Boolean data types.
- FALSE - Represents the FALSE value condition for Boolean data types.
- ALFALENG - Represents the length of a character string that may be packed into a single word. For PASCAL, ALFALENG = 2.

The general form for equating an identifier to a constant value is:

identifier = constant-value

The general form of the constant definition part of the D/D segment is:

CONST identifier = constant-value, /identifier = constant-value,\ ...

Example:

```
CONST  MAXCOR = $7FFF,
       TOTLIN = 32,
       EMPTY = NIL,
       MINUS1 = -1,
       LINES = TOTLIN;
```

Note that the last constant definition is followed by a semicolon instead of a comma.
(An absolute entry point (ENT) is generated for each constant identifier.)

2.3.3 TYPE DEFINITIONS

Data values are represented by storage elements referred to as variables. Every data element in PASCAL has a type definition associated with it. The data type essentially defines the set of values that a data element may assume. A data type may be directly described in the variable declaration, or it may be referenced by a type identifier. The type definition part of the D/D segment assigns type identifiers to explicit type definitions.

Note that a type definition does not generate data, but establishes an identifier to represent the defined data type in any subsequent declaration of variable data or function type.

The general form of the type definition part of the D/D segment is:

TYPE type-definition; /type-definition;\...

The general form for equating a type identifier to a data type is:

type-identifier = data-type-definition,

The data type may be specified as any one of the following:

1. A scalar type
2. A structured type
3. A pointer type

2.3.3.1 SCALAR

The basic PASCAL data types are the scalar types. Their definition indicates a distinct and ordered set of values. A scalar-type description introduces the type identifier and a list of constant identifiers that it represents. The general form of the scalar-type definition appears,

type-identifier = (constant-identifier /, constant-identifier\...)

A scalar-type definition of the above form is also referred to as an enumeration type.

Examples:

MONTH = (JAN, FEB, MARCH, APRIL, MAY, JUNE, JULY, AUG, SEPT, OCT, NOV, DEC)

SUIT = (DIAMOND, HEART, SPADE, CLUB)

BINTYP = (DATABIN, TAGBIN, DISKBN)

COLOR = (RED, YELLOW, BLUE, GREEN)

CODES = (ASCII, BAUDOT, EBCDIC, XCESS3)

Data variables that are declared to be scalar may contain any of the constant values represented by the constant identifier list.

Certain scalar types are predefined by the PASCAL compiler. These standard scalar types include Boolean, integer, and CHAR (character).

BOOLEAN TYPE

The Boolean type denotes the pair of truth values whose identifiers are TRUE and FALSE. The Boolean predefinition is equivalent to the following scalar-type definition:

```
BOOLEAN = (FALSE, TRUE)
```

INTEGER TYPE

The integer type represents the set of whole numbers. It represents the range of integer values that can be specified within a single, central memory word.

$$-32767 \leq \text{integer} \leq 32767$$

If we consider the literal representation of the integer values as a special form of a constant identifier, the integer-type predefinition may be considered to be equivalent to the following scalar-type definitions:

```
INTEGER = (-32767, -32766, . . . , -0, 0, . . . , 32767);
INTEGER = ($8000, $8001, . . . , $FFFF, $0, $1, . . . , $7FFF);
```

CHAR TYPE

The CHAR (character) type represents the entire set of graphic characters that are defined by the CDC 63-character set. This code is presented in Table 2-1 together with its associated ordinal valu (the hexadecimal numbers that represent the characters).

TABLE 2-1. PASCAL GRAPHICS

Low-Order Ordinal Hexadecimal Digit	High-Order Ordinal Hexadecimal Digit				
	2	3	4	5
0	blank	0	≤	P
1	∨	1	A	Q
2	≠	2	B	R
3	≡	3	C	S
4	$	4	D	T
5	reserved	5	E	U
6	∧	6	F	V
7	↑	7	G	W
8	(	8	H	X
9	)	9	I	Y
A	*	:	J	Z
B	+	;	K	[
C	,	<	L	≥
D	-	=	M	]
E	.	>	N	¬
F	/	↓	O	⌐

Within memory, a character appears right-justified in a 16-bit word with leading zeros.

If we consider the literal representation of the character constants as a special form of constant identifiers, and if their ordinal value implies the character ordering, then the CHAR type predefinition may be considered to be equivalent to the following scalar-type definition:

CHAR = (≡ ≡, ≡∨≡, . . . , ≡ ⌐→≡);

The ordered set of characters as they appear in the preceding table.

SUBRANGE TYPE

A specific interval (subrange) of a known integer or character type may be defined as a distinctive scalar type. This subrange type is defined as:

subrange-identifier = minimum-constant .. maximum-constant

An enumeration-type subrange may not be specified in PASCAL.

Examples:

```
BINCNT = 0..127;
LETTRS = ≡A≡..≡Z≡;
PAGADR = $0..$7F;
ONEBIT = 0..1;
```

Note that the standard definition, integer, is in reality a subrange of the whole numbers:

```
INTEGER = -32767..32767
```

2.3.3.2 STRUCTURED TYPE

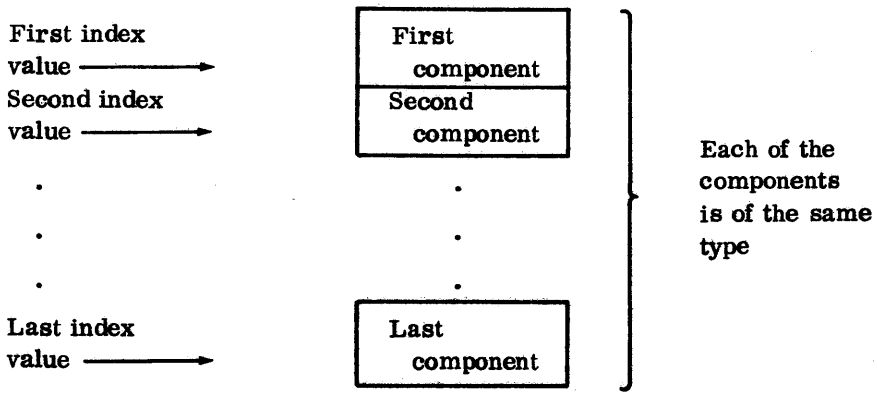
A data type may be defined in terms of previously defined constituent types. A data type that consists of several related variables (components) is said to be a structured type. A structured type is defined by describing the types of its components and indicating its structuring method. Three methods of structuring are available in PASCAL, each distinguished by the manner in which individual components are accessed: array structures, record structures, and set structures.

ARRAY STRUCTURE TYPE

A variable that is typed as an array structure consists of a fixed number of repeated components that are all of the same type, the component type. An array structure is characterized by the following:

1. Each individual component of the array is directly accessible by an assigned index into the array.
2. The number of its components is defined when the array variable is introduced and remains unchanged thereafter.

An array structure is shown as:



The general form of a single-dimensioned array-type definition is:

type-identifier = ARRAY [index-type] OF component-type

Where: index-type is the range of all possible indices.

component-type is any type, those described as well as those to be described.

The size of an array variable may be determined by the count of components (implicitly known from the index type) and the size of each component (determinable from the component type).

Examples:

CNTTBL = ARRAY [1..100] OF INTEGER

CHRARY = ARRAY [1..58] OF LETTRS

ODDBAL = ARRAY [COLOR] OF BOOLEAN

Note that with the last example if a variable is typed with the type identifier ODDBAL and further, if COLOR is defined as a scalar list of constant identifiers as in Section 2.3.3.1, then the ODDBAL variable will consist of four components indexed as [RED], [YELLOW], [BLUE], and [GREEN], where each component may be specified as TRUE or FALSE.

It is possible to define an array whose component type is also an array.

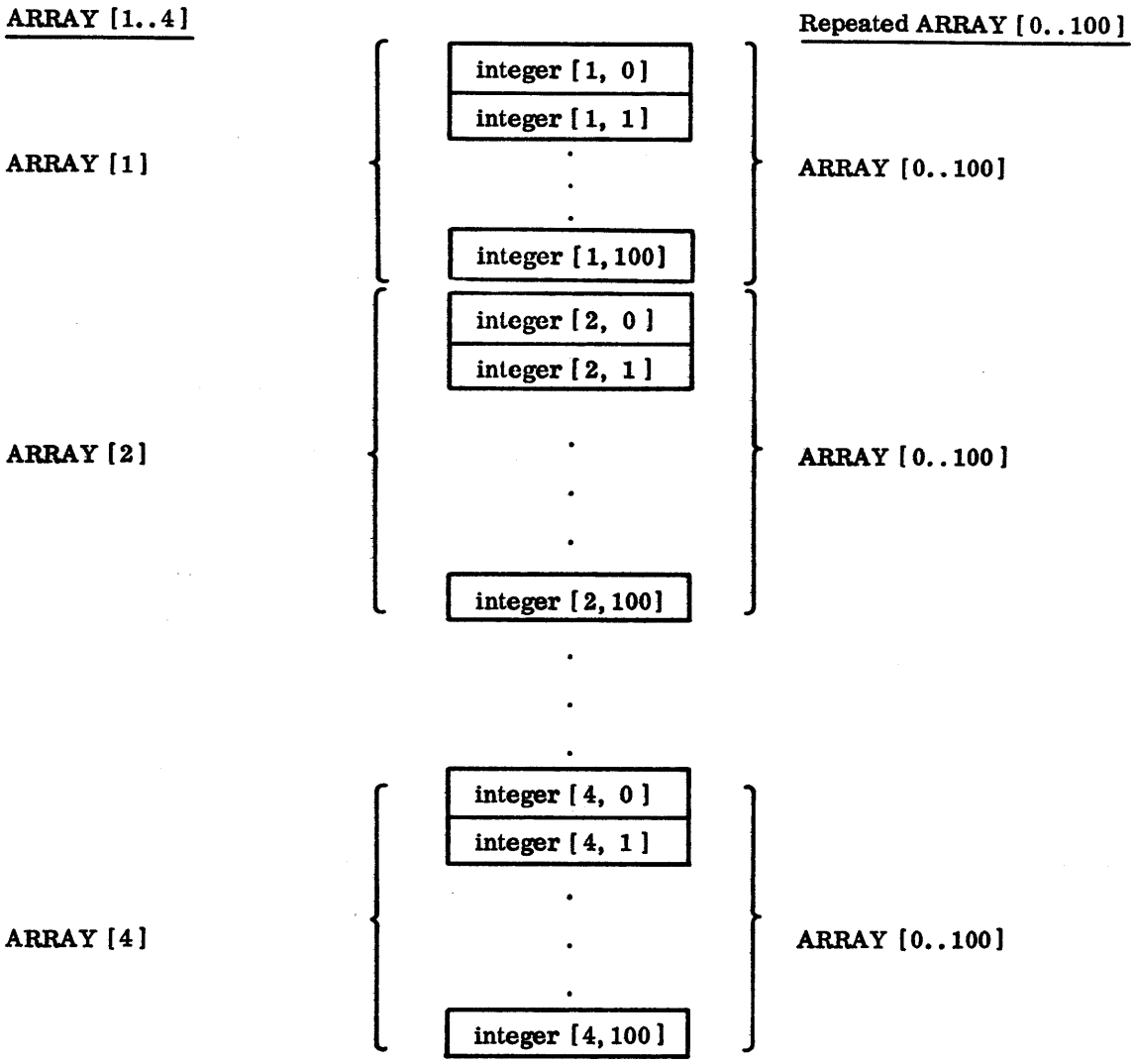
An example declaration might appear,

ARRAY [1..4] OF ARRAY [0..100] OF INTEGER

This type of a structure is called multidimensional and may be represented in a more convenient manner:

ARRAY [1..4, 0..100] OF INTEGER

Pictorially, the two-dimensional array in the above example would appear in the following form:



The total number of integer components that comprise this two-dimensional example equals:

```
  4   (Number of components in first dimension)
x 101 (Number of components in second dimension)
---
404   (Total integer components)
```

The concept of defining ARRAYs of ARRAYs, etc., may be extended to three or more dimensions.

The general form of an array-type definition, including multidimensionality, is:

type-identifier = ARRAY [index-type /, index-type\...] OF component-type

RECORD STRUCTURE TYPE

A variable that is typed as a record structure consists of a set of components (fields) that are not necessarily of the same type. A record structure is characterized by the following:

1. Each field of the record is directly accessible by qualifying the record variable name with the desired field name. (See Section 2.4.1.)
2. A portion of the record, if not all, may be defined to have more than one set of field definitions occupying the same area. Such a multidefined (multipurpose) area is called a variant.
3. The size of each record is determined by the worst-case (largest size) arrangement of its variants.

A record-type definition contains two basic parts: fixed and variant. Either one or both may be present in a definition; however, if both are present the fixed part must precede the variant part. The general form of a record type definition is:

```
type-identifier = RECORD
                    /fixed-part \
                    /variant-part \
                  END
```

The fixed part introduces field names associated with the record and assigns a type to each of the introduced fields. The general form of the fixed part is:

```
field-identifier /,field-identifier\ ...: type-definition
/;field-identifier /,field-identifier\...: type-definition\...
```

Examples:

```
DATE = RECORD
          DAY: 1..31;
          MONTH: 1..12;
          YEAR: INTEGER;
          LEAP: BOOLEAN
       END

BIN = RECORD
          FSTCHR: 1..58;
          LSTCHR: 1..58;
          SOMBN, EOMBN: BOOLEAN;
          TEXT: ARRAY [1..58] OF CHAR;
          CHAIN: ↑BIN
      END
```

(Refer to section 2.3.3.3 for description of pointer types, e.g., ↑ BIN.)

The type definition may also be a previously defined type identifier. For example, the BIN type definition could also be specified in the following manner:

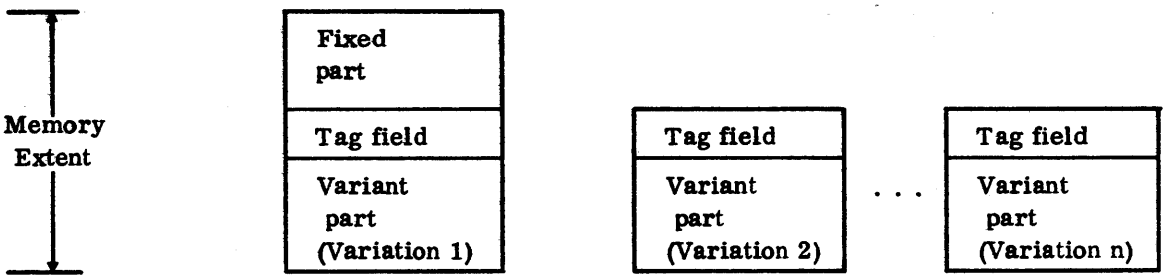
```
TXTYP = ARRAY [1..58] OF CHAR;
BIN = RECORD
        FSTCHR: 1..58;
        LSTCHR: 1..58;
        SOMBN, EOMBN: BOOLEAN;
        TEXT: TXTYP;
        CHAIN: ↑BIN
      END
```

The variant part may in turn contain a fixed part, a nested variant part, or both. The fixed part is specified in the form described above. The nested variant part is similar to the variant part, but the deepest nested variant part must contain only a fixed part. The general form of the variant part is:

```
CASE tag-field: tag-type-identifier OF
     case-label /,case-label\ ...:    ( /fixed-part\/variant-part\)
     /;case-label /,case-label\ ...: ( /fixed-part\/variant-part\) ...
```

The tag field identifies the first field assigned to the variant part and is common to each of the variations to be defined. The tag field must be declared a scalar type via a type identifier. The case label is a constant value that may be assigned to the tag field. It is used to associate a variation in the structure with a particular tag-field value.

A variable record structure is shown:



The variant part is depicted in the same manner as a variable record.

PASCAL provides an option wherein the tag field of a variant record structure occupies no memory space; however, it must be specified in the type definition in order to assign case labels to variants. An options comment is used to specify this option.

Examples:

The following record definition includes both a fixed and a variant part:

```
TAG = RECORD
        MTFCHN: ↑TAG;
        CODSET: CODES;
        MSGLNG: INTEGER;
      CASE BTYP: BINTYP OF
        TAGBIN: (PRIORT: 1..4;
                 NODATA: BOOLEAN;
                 TRANSP: BOOLEAN);
        DISKBN: (DSKADR: INTEGER;
                 DATAB: BOOLEAN)
      END
```

This is a more complex example of nested variant parts:

```
BEING = (ADULT, CHILD)
SEX = (MALE, FEMALE)
PERSON = RECORD
           NAME: ARRAY [1..10] OF CHAR;
           AGE: INTEGER;
         CASE HUMAN: BEING OF
           ADULT:   (CASE ASEX: SEX OF
                     MALE: (FATHER: BOOLEAN);
                     FEMALE: (MOTHER: BOOLEAN));
           CHILD: (CSEX: SEX)
         END
```

The following three cases are the variations defined by the PERSON type identifier:

NAME
AGE
HUMAN
ASEX
FATHER

NAME
AGE
HUMAN
ASEX
MOTHER

NAME
AGE
HUMAN
CSEX
////////

SET TYPE

A set-type definition assigns a set identifier and associates it with a particular scalar type. A set-type variable represents a set or subset of the associated scalar values. A set variable will be assigned sets of values during the action portion of a procedure. The general form of the set-type definition is:

```
type-identifier = SET OF scalar-type;
```

PASCAL permits a maximum of 16 elements in a set. Within PASCAL a set of items occupies a full word, where each element is associated with a bit within the word. The element assignment is ordered from the rightmost bit.

Examples:

```
HUE = SET OF COLOR

SEASON = SET OF MONTH
```

Note that scalar types associated with sets are limited to the integer subrange types where the lower bound is greater than or equal to 0 and the upper bound is less than or equal to 15, and enumeration types with 16 or fewer elements. (The CHAR scalar type may also be used, but it is limited to the special characters $00 through $0F when associated with sets.)

PACKED STRUCTURES

The PASCAL language provides a specialized keyword (PACKED) which, when specified, indicates to the compiler that the associated data components should be arranged in a compact form whenever possible. When PACKED is specified, it immediately precedes the structure definition's primary keyword:

```
PACKED ARRAY

PACKED RECORD
```

The presence of PACKED has no effect on the interpretation of the program. Its use is considered when economy of storage is more important than efficiency of access.

PACKED ARRAY will have significance to the PASCAL compiler only for arrays that consist of CHAR-type components. An unpacked array with CHAR-type components is interpreted as an array of characters, one character per word. A packed array with CHAR-type components defines an array of characters, two characters per word. Packed arrays with components of any other type will be allocated space in the same manner as an unpacked array.

A packed record, as interpreted by PASCAL, will compact scalar-type fields according to the following rules:

1. Integers will occupy a full word.
2. An integer subrange type will occupy the smallest portion of a word that will retain all values ascribed to the subrange. The assigned field position will begin:
   a. At the next available bit position if there is sufficient space in the word.
   b. At the first bit position of the next available word if there is insufficient space in the current word.
3. A scalar enumeration type will occupy the smallest portion of a word that will retain all values ascribed to the scalar type. The field position assignments will be made in the same manner as for the subrange type. The first constant of the enumeration list will be assigned a zero value, the second constant a one value, etc.

4. A Boolean type will occupy a single bit and will be assigned to the next available bit position type:

   0 = False
   1 = True

5. A CHAR-type will occupy an eight-bit field beginning at the next available bit position in the word. If there is insufficient space to retain the entire character in the current word being filled, it will be assigned to the leftmost byte of the following word.

All other field types associated with the packed record will be assigned space in the same manner as an unpacked record.

When a field ends in the middle of a word and the next field specified cannot fit in the remainder of that word, the initial field is expanded to the end of the word. An exception is the case of tag fields (or, if no space is allocated for tag fields, the last field of the fixed part), which are not expanded.

2.3.3.3 POINTER TYPE

Variables for a program may be declared in the variables declaration portion of the D/D segment. Declared variables are predefined and local to the procedure; consequently, they are considered as static variables. Variables may also be generated dynamically by calling a standard procedure, NEW, which returns a pointer to the allocated data.

A pointer-type definition assigns a pointer identifier and associates it with (binds it to) another defined type. A pointer-type variable is the storage element that points to a dynamically allocated variable. The allocated variable is of the pointer's bounded type. The general form of the pointer type definition is:

type-identifier = ↑defined-type-identifier

Examples:

IPTR = ↑INTEGER

TAGTYP = ↑TAG

PERTYP = ↑PERSON

CPTRTY = ↑CNTTBL

BPTR = ↑BIN

2.3.4 VARIABLE DECLARATIONS

The variable declaration part of the D/D segment lists each of the variables that is local to the procedure. Each local variable is assigned an identifier and bound to a type. The general form of the variable declaration part of the D/D segment is:

VAR variable declaration; / variable declaration; \ ...

The general form for a variable declaration is:

variable-identifier /, variable-identifier \ ... : type-definition

where the type definition may be a type identifier or an explicit type definition.

Example:

```
VAR
    I, J: INTEGER;
    CARD: SUIT;
    NODMND: SUIT;
    LCNT: 0..15;
    DIGITS: SET OF 0..15;
    P, Q: BOOLEAN;
    BINPTR: ↑BIN;
    MEMBRS: ARRAY [1..TOTLIN] OF PERSON;
    ORANGE: HUE;
    STATE: ARRAY [0..15] OF BOOLEAN;
    ALETTR: LETTRS;
    TITLE: ARRAY [1..10] OF LETTRS;
    IAM: BIN;
    TMPBNS: ARRAY [1..2] OF BINS;
    PAGPTR: ↑PAGADR;
    TINT: COLOR;
    MO: SEASON;
    MON: SEASON;
```

In PASCAL, storage areas for local variables may be assigned in-line at compile time, or they may be allocated dynamically during run-time immediately after a called procedure is entered. The programmer selects the particular method used, specifying the desired option in an options comment. (refer to Appendix B).

Dynamic allocation of local variable space permits the compiler to generate recursive code. In-line generation of local variables prohibits recursive code generation.

In-line variables will be assigned memory space in the same order as the variable declarations are specified. With a priori knowledge as to the manner in which storage elements are assigned, a programmer may predictably structure (on a relative basis) a local variable area.

A program's global area (the main procedure's local variable area) may generate only in-line variables. PASCAL recognizes this global area uniquely.

An assembly language routine may access a variable in the global area by using the following external definitions:

1. A relocatable external address for each variable identifier appearing in the VAR part of the D/D segment.
2. An absolute external displacement for each field identifier appearing in the TYPE definition part of the D/D segment.

3. An absolute external field position for each field identifier appearing in the TYPE definition part.

Note that within the definition of the PASCAL language it is possible to specify separate record types that have like field identifiers. It is the responsibility of the programmer to avoid duplication of global field names in different type definitions to prevent the generation of ambiguous entry definitions.

2.3.5 VALUE INITIALIZATION

PASCAL provides compile time value initialization for global variables and local variables in procedures that were compiled selecting the in-line variables option. The value initialization part of the D/D segment lists selected variables and assigns them initial value constants. The general form of the value initialization part is:

VALUE a value assign; /a value assign;\...

Variables may be initialized only if they have been previously declared in the variable declaration part. Variables that may be initialized are restricted to the following types:

1. Scalar data types
2. Single dimensioned arrays whose components are of an unpacked scalar type
3. Single dimensioned char-type packed arrays
4. Packed/unpacked record as an unpacked array of integers

A value assignment appears in four general formats.

SINGLE VALUED SCALARS

variable-identifier = initial-value-constant

Example:

```
VALUE

    I = 5;
    CARD = HEART;
    NODMND = HEART;
    P = FALSE;
    ALETTR = ≡D≡;
```

ARRAY OF SCALARS

variable-identifier = (constant /,constant\...)

Example:

```
VALUE
    TITLE =  (≡P≡,≡A≡,≡S≡,≡C≡,≡A≡,≡L≡);
```

The value of an uninitialized variable is undefined. In the above example, if TITLE is declared as in the example in Section 2.3.4, then the last four characters of TITLE are undefined.

Within the value initialization, the sequence of repeating constants can be abbreviated by preceding the constant that is to be repeated with an unsigned repetition factor followed by an asterisk (*). For example, if it is desired to complete the initialization of the TITLE array by defining the last four characters as blanks, then the specification of the TITLE initialization could appear,

```
TITLE =    (≡P≡, ≡A≡, ≡S≡, ≡C≡, ≡A≡, ≡L≡, 4*≡≡)
```

PACKED ARRAY OF CHARACTERS

variable-identifier = (character-string-constant)

A char-type packed array may be initialized by specifying the initial value as a character string constant.

Examples:

```
VAR
    MESSAGE: PACKED ARRAY [1..9] OF CHAR;
VALUE
    MESSAGE = (≡A MESSAGE≡);
```

PACKED/UNPACKED RECORD

variable-identifier = (constant /, constant\ ...)

A packed/unpacked record is treated as an array of integers so that each integer constant specified in the value list is assumed to represent a full word in the record, regardless of its actual field type or the number of fields ascribed to any one word. The number of integer constants that may be specified in the list must not exceed the total number of words assigned by the compiler to the record.

Example:

```
TYPE
     RECTYP = PACKED RECORD
                  FIELD1: 0..$7FFF;
                  FIELD2: BOOLEAN;
                  FIELD3: INTEGER
              END;
VAR
     REC: RECTYP;
VALUE
     REC = ($273D, 47);
```

In the above example the initial values assigned to the fields would be:

```
FIELD1 = $139E
FIELD2 = TRUE
FIELD3 = 47
```

2.4 ACTION SEGMENT

A procedure provides a functional requirement for a program. Execution of a procedure usually involves the assignment of a determined value(s) (either by calculation or logical consequence) to a global variable and/or a variable parameter. The action segment of a PASCAL procedure defines the logical path of processing to be performed in determining the required values.

2.4.1 VARIABLE SPECIFICATIONS

A variable may be designated in the body of a procedure's action segment in four ways:

- As an entire variable
- As a pointer variable to a scalar type
- As an array variable
- As a field variable

An entire variable is represented by a simple specification of the variable identifier. Entire variable specifications are restricted to the following types, which do not themselves appear in structured types:

- Scalars
- Sets
- Pointers

A pointer variable to a scalar type represents a dynamically allocated scalar variable. Since there is no unique identifier to represent the allocated variable, it is specified by suffixing an up-arrow (↑) to the identifier declared for the pointer variable. For example,

```
PAGPTR↑
```

An array variable is specified by suffixing an appropriate index value enclosed in brackets to the array's declared identifier. For example,

```
TITLE[2]
STATE[LCNT]
MSG[BINO, CHIDX]
```

It is possible to specify the index in terms of an expression as long as the result is of the index type. For example,

```
STATE[LCNT + 2]
```

A field variable is specified by denoting the record variable followed by a period (.) followed by the field identifier. The field identifier is said to be a qualification of the record variable. A record variable may appear as:

- A record identifier
- An indexed array identifier of type RECORD
- A pointer variable to a RECORD structure
- A field variable that is itself a record variable

Examples:

```
IAM.FSTCHR
TMPBNS[1].EOMBN
MEMBRS[30].HUSBND
```

A pointer-type record variable to a record structure is denoted by specifying the pointer identifier suffixed by an up-arrow (↑). For example,

```
BINPTR↑.TEXT[FSTCHR]
```

It is possible to specify a field variable that is itself a record variable. In the example in section 2.3.3.2, a TYPE record structure identified as BIN includes a field name, CHAIN, that is typed as a pointer to the BIN record structure. The CHAIN pointer essentially introduces a level of indirectness. An indirect specification of this type might appear as:

```
BINPTR↑.CHAIN↑.EOMBN
```

Specifying a pointer variable (e.g., CHAIN without the suffixed ↑) denotes the pointer value; whereas adding the ↑ suffix denotes the variable referenced by the pointer variable.

A procedure may reference any of its local variable identifiers as well as those variable identifiers that are declared global to it. All others, including those declared in its local procedures, may not be referenced.

2.4.2 EXPRESSIONS

Expressions are constructs that direct the execution of a computation; the result is the value of the computation. An expression is specified within PASCAL as:

- A standalone value, optionally preceded by a sign
- A simple expression involving values of like type separated by an operator
- A compound expression involving parenthesized (possibly understood) expressions representing values of like type separated by an operator.

A value appearing in an expression, sometimes called an operand, may be presented as a variable, constant, set, function, or expression. A standalone value represents the result of the expression. The value type is the result type. The negation of an integer or Boolean value may be represented by the following method:

```
-    integer-value
¬    Boolean-value
NOT  Boolean-value
```

The keyword NOT and ¬ are equivalent symbols. A plus sign (+) preceding an integer value is understood, but it may be specified. The following are examples of standalone expressions:

```
4
P
TITLE [4]
PAGPTR
NOT BINPTR↑.EOMBN
+LCNT
-15
```

Simple and compound expressions involve operands and operators. The PASCAL language defines a fixed set of operators that are associated with values of specific data types. The available operators may be subdivided into the following groups:

1. Arithmetic
2. Relational
3. Set
4. Boolean

2.4.2.1 ARITHMETIC OPERATORS

The arithmetic operators are restricted to use with operands that are of the integer type or a subrange thereof. The result of a computation involving arithmetic operators is itself an integer type. The following lists the arithmetic operators, a formal specification of the operators in a simple expression, and the implied operation:

Operator	Simple Expression	Implied Operation
+	$Val_1 + Val_2$	Addition
-	$Val_1 - Val_2$	Subtraction
*	$Val_1 * Val_2$	Multiplication
DIV	Val_1 DIV Val_2	Division, Val_1 divided by Val_2
MOD	Val_1 MOD Val_2	Modulo, remainder of Val_1 divided by Val_2

An expression that involves arithmetic operators and results in an integer type is called an arithmetic expression.

Examples:

```
LCNT + 4

4 * LCNT

(LCNT DIV 4) MOD 3
```

Note that in some cases an arithmetic expression may be added to or subtracted from a pointer variable (see Section 3.1).

2.4.2.2 RELATIONAL OPERATORS

The relational operators specify a comparison between their two associated operands, the result of which is always of Boolean type. The relational operators that may be specified depend on the operand types involved in the comparison. A relational expression may be categorized into operations involving operands of:

1. Scalar type
2. Set type
3. All types

The scalar type defines an ordered set of values for which the following comparisons may be made:

Operator	Simple Expression	Operation
=	$Val_1 = Val_2$	Equality
≠	$Val_1 \neq Val_2$	Inequality
<	$Val_1 < Val_2$	Val_1 less than Val_2
>	$Val_1 > Val_2$	Val_1 greater than Val_2
≤	$Val_1 \leq Val_2$	Val_1 less than or equal to Val_2
≥	$Val_1 \geq Val_2$	Val_1 greater than or equal to Val_2

Examples:

```
LCNT = 4

4 ≤ LCNT

NODMND > HEART
```

The result of an arithmetic expression is itself a scalar value that may be specified as a scalar operand. Expressions involving both arithmetic and relational operators are computed by resolving all of the arithmetic operations first and then resolving the relational operations. The arithmetic operators are of higher precedence than the relational operators. The following is an example of this expression type:

```
LCNT + 4 < 15
```

Variables that are of set type may be tested for equality, inequality, and inclusion. In addition, a scalar value may be tested for membership in a set variable whose base type is the same as that of the scalar value.

The relational operators that involve set operands are listed in the following table:

Operator	Simple Expression	Operation
=	$Set_1 = Set_2$	Equality
≠	$Set_1 \neq Set_2$	Inequality
≤	$Set_1 \leq Set_2$	Inclusion; i.e., set_1 included in set_2
≥	$Set_1 \geq Set_2$	Inclusion; i.e., set_2 included in set_1
IN	Val IN Set	Membership

The relational operators <, ≤, ≥, > may also be applied to packed arrays with components of type char.

≤ and ≥ denote the set functions of inclusions normally represented as $\subseteq$ and $\supseteq$.

Examples:

```
4 IN DIGIT
MO ≤ MON
```

All other variable types may be compared for equality/inequality. Both operands must, however, be of the same type.

2.4.2.3 SET OPERATORS

Variables that are of the same set type may be operated upon and derive a result that is of the same set type. The set operators available and their operations are listed in the following table:

Operator	Simple Expression	Operation
∨	$Set_1 \vee Set_2$	Set union
∧	$Set_1 \wedge Set_2$	Set intersection
+	$Set_1 + Set_2$	Exclusive OR of sets
-	$Set_1 - Set_2$	Set difference

2.4.2.4 BOOLEAN OPERATORS

Boolean operators indicate operations on Boolean values, which in turn result in a Boolean value; the so-called truth value. Boolean operations include logical AND and logical OR. In addition, a Boolean expression may be prefaced with the logical NOT, where a Boolean operand includes:

- A Boolean constant: TRUE or FALSE
- A Boolean variable
- A Boolean expression
- A relational expression

The Boolean operators available are listed in the following table:

Operator	Simple Expression	Operation
∧	$Bool_1 \wedge Bool_2$	Logical AND
∨	$Bool_1 \vee Bool_2$	Logical OR
¬	¬ Boolean expression	Logical negation of Boolean expression
NOT	NOT Boolean expression	Logical negation of Boolean expression

Expressions whose results are Boolean are called Boolean expressions. These include expressions that involve relational operators as well as Boolean operators.

2.4.2.5 OPERATOR PRECEDENCE

Operators are applied in the order of their precedence. Operators with higher precedence are applied first. Operators that have the same precedence are applied from left to right.

Precedence	Operator
4	¬ NOT
3	* DIV MOD ∧
2	+ - ∨
1	= ≠ < ≤ > ≥ IN

2.5 FORWARD REFERENCE DECLARATION

A procedure or function procedure may be referenced before it is declared. The forward reference declaration is used when the forward-referenced routine is a function procedure or a procedure of level 2 or greater. The format of this forward reference is identical to the called procedure's heading information with the addition of the suffixed keyword FORWARD. Its general form is:

PROCEDURE proc-identifier / (formal-parameters) ; FORWARD

Similarly, the general form for forward-referencing a function procedure is:

FUNCTION function-identifier /(formal-parameters)\ :type; FORWARD

The prestatement of a procedure or function procedure calling sequence must immediately precede the heading information for the procedure or function procedure that is about to be declared. Note that a forward reference applies to all subsequent procedure declarations that forward-reference the same procedure, so the prestatement does not have to be repeated. It is not necessary to repeat the formal parameter section in the declared procedure's heading information.

Example:

```
PROCEDURE  P (VAR X:INTEGER); FORWARD;
FUNCTION   Q (Y:INTEGER): INTEGER; FORWARD;
PROCEDURE  R;
     VAR  Z:INTEGER;
     BEGIN
         Z :=1; P(Z); Z :=+Q(Z)
     END;
PROCEDURE  P;
     BEGIN  X :=X+1 END;
FUNCTION   Q;
     BEGIN  Q:=Y + 1 END;
```

2.6 COMMENTS

Comments may be introduced in a procedure at any position that does not violate a keyword or an identifier. The text of a comment may include all graphic characters except ↦ and ↓. The $ character may not be the first character in the text. The two characters ↦ and ↓ serve to delimit the text of a comment; the character ↦ indicates the beginning of a comment and the character ↓ indicates the end of the comment.

Note that a comment is a delimiter for keywords and identifiers within the source program.

PASCAL STATEMENTS AND STATEMENT LABELS 3

The action segment is specified by a list of executable statements, each of which denotes an algorithmic action to be performed. The general format of a statement is:

/label:\ statement;

Any statement may be labeled with an unsigned integer. The statement label identifies a particular statement for use with the GOTO statement. For PASCAL, the statement label may be assigned within the limits

$1 \leq$ statement-label ≤ 9999

The constituents of simple statements are constants, variable identifiers, procedure/function identifiers, keywords, and special characters. The symbol arrangements (syntax structure) associated with each simple statement and the description of its action are presented in this section. Note that constants, identifiers, and keywords are delimited by one or more blank characters or by a special character. Special characters are self-delimiting and need not be surrounded by blanks.

A statement may be specified in two forms: as a simple statement and as a composite statement. A simple statement refers to a basic action type that may be specified in the PASCAL language. A composite statement comprises a sequence of statements (which themselves may be simple and/or composite) that are to be executed in the same sequence as they are specified.

The different types of simple action statements thay may be specified in PASCAL are described in this section. The sequence of statements that comprises a composite statement is delimited by the statement brackets BEGIN and END. The general form of a composite statement is:

/ label:\ BEGIN statement/;statement\... END

Note that references to statement imply either a simple or composite statement.

3.1 ASSIGNMENT STATEMENT

The assignment statement replaces the current value of a variable with the result of an expression. The result of the expression must be the same type as the receiving variable. Its general form is:

receiving-variable := expression

The receiving variable in this case may be either a scalar identifier, a set identifier, or a function identifier.

The assignment statement also serves to equate variables of like type:

```
destination-variable := source-variable
```

The value of the source variable is copied into the destination variable. A source variable, which is an expression consisting of a standalone value, represents a special case of the previous assignment definition; however, this second definition is particularly meaningful for copying pointer or structured variables.

Lastly, the assignment statement serves to assign a set of constants to a set variable. The constants may be specified as constant or as constant identifiers, but they must be of the same type as the set variable to which they are assigned. The general form for this assignment is:

```
set-variable := [/constant/, constant\...\]
```

It is possible to have an empty set for the set variable. In this event the enclosing brackets contain no constants: [].

Examples:

```
P := TRUE
Q := P
CARD := CLUB
I := J DIV 4+(I * LCNT)
BINPTR .CHAIN := BINPTR
DIGIT := [0,1,2,3,4,5,6,7,8,9]
```

PASCAL is not sensitive to the variable types associated with a pointer variable when the pointer variable (a reference without the suffixed ↑) appears in an assignment statement. This means that the memory address appearing in a pointer variable may be assigned to another pointer variable regardless of their associated types. For example,

```
VAR
     P1 : ↑INTEGER;
     P2 : ↑BOOLEAN;
BEGIN
     NEW (P1):
     P2 := P1
END
```

PASCAL also permits an integer value to be added to or subtracted from a pointer variable within an assignment statement when specified in the following general forms:

```
pointer-variable  :=  pointer-variable ± arithmetic-expression
pointer-variable  :=  NIL ± arithmetic-expression
```

```
VAR
      P1 : ↑INTEGER;
      P2 : ↑BOOLEAN;
BEGIN
      NEW (P1);
      P1 := P1 + 2;
      P1↑ := 3;
      P2 := P1 + (4 - P1↑)
END
```

3.2 GOTO STATEMENT

The GOTO statement transfers control to the statement with the given label. The GOTO specification has two general forms:

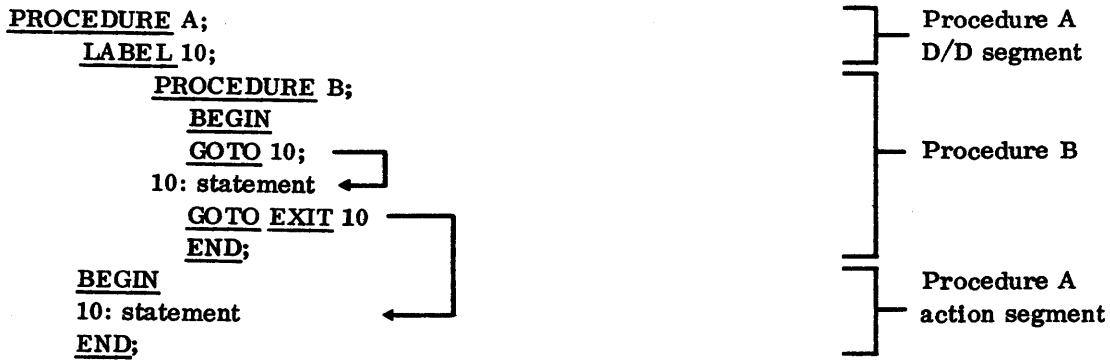
GOTO statement-label

GOTO EXIT statement-label

The scope of a label is the procedure within which it is defined; that is, it is not possible to jump (transfer control) into a local procedure.

The first form of the GOTO statement is used for jumps to statements that appear in the procedure, whereas the second form is used to jump outside of the procedure (i.e., to a statement within a procedure that is global to the one where the GOTO is specified). In this second case the statement label must be specified in a label declaration of the global procedure.

The following example depicts the GOTO relationship described above:



3.3 IF STATEMENT

The IF statement specifies conditional execution of its component statements, depending on the outcome of an associated Boolean expression. The general form of an IF statement specification is:

```
IF boolean-expression THEN statement /ELSE statement\
```

The Boolean expression is evaluated; if the result is true, the statement following THEN is executed before proceeding to the next statement. If the result is false, then

1. The statement following the ELSE is executed, if ELSE has been specified.
2. Control transfers immediately to the statement following the IF, if ELSE was not specified.

Examples:

```
IF P THEN LCNT := 5

IF P ∧ Q THEN LCNT := I
         ELSE BEGIN
                ALETTR :=≡Z≡;
                I := J;
              END
```

This composite statement constitutes the statement following the ELSE.

```
IF CARD = HEART THEN CARD := CLUB
                ELSE CARD := HEART
```

3.4 CASE STATEMENT

The CASE statement consists of an expression (the selector) and a list of statements, each labeled by one or more constants of the type of the selector. It specifies that one statement is to be executed whose constant label is currently equal to the value of the selector. If the selector value does not equal any of the specified labels, then processing will proceed with the statement following the CASE statement. The general form of the CASE statement is:

```
CASE expression OF
     constant /, constant\...: statement
     /;constant /, constant\...: statement
                    .
                    .
                    .
END
```

Examples:

```
CASE I OF
     1: J:=I;
     2: J:=I*I;
     3: J:=I*I*I
END

CASE CARD OF
     HEART: ¬AM.TEXT [¬AM.FSTCHR] := ≡A≡;
     CLUB:  BEGIN P := TRUE;
               Q := FALSE
            END
END
```

3.5 WHILE STATEMENT

The WHILE statement specifies that an associated statement may be executed repeatedly until a controller Boolean expression becomes false. If the Boolean is initially false, the associated statement will not be executed. The general form of the WHILE statement is:

WHILE boolean-expression DO statement

The repeated statement must affect a variable included in the Boolean expression that will at some repetition result in a false value, or it must exit the WHILE with a GOTO statement; otherwise this statement would repeat indefinitely.

Examples:

```
WHILE I<10 DO
     BEGIN
          J := J*J;
          I := I+1
     END

WHILE ¬ ¬AM. TEXT [I] IN DIGITD DO
     IF I>58
          THEN GOTO 4
          ELSE I := I+1
```

3.6 REPEAT-UNTIL STATEMENT

The REPEAT-UNTIL statement provides an alternate form for specifying that an associated list of statements be executed repeatedly until a controlling Boolean expression becomes true. It differs from the WHILE statement in that,

1. A list of statements may be repeatedly executed, as opposed to a single statement.
2. The test of the controlling Boolean expression follows the list of statements to be executed repeatedly. This implies that the statement list will be executed at least once.

The general form of the REPEAT-UNTIL statement is:

REPEAT statement /;statement\... UNTIL boolean-expression

The list of statements being executed repeatedly must affect a variable included in the Boolean expression that will at some repetition result in a true value, or the repeated statements must terminate by exiting via a GOTO statement. If this is not done, the list of statements will be repeated indefinitely.

Examples:

```
REPEAT I:=J;
       J:=J+4
UNTIL I>$1000

ORANGE := [RED, YELLOW]
REPEAT TINT := SUCC (TINT)
UNTIL ┌─ TINT IN ORANGE
```

3.7 FOR STATEMENT

The FOR statement indicates that a statement is to be executed repeatedly while a progression of values is assigned to a control variable. It may be specified in two basic forms:

FOR control-variable := initial-expression
    TO final-expression DO statement

FOR control-variable := initial-expression
    DOWNTO final-expression DO statement

The first form is used to assign values to the control variable in increasing order, while the second form will assign values in a decreasing order.

The control variable must be a scalar type. The initial and final expressions must yield a value of the type for which the control variable is defined. The control variable is assigned all of the values that lie in the range delimited by the initial and final values. The associated statement will be executed once for each assignment.

The control variable is available for reference within the associated statement, but it may not have its value changed. In addition, any variables appearing in the final expression may not be changed. The control variable should be considered an unknown quantity after the FOR statement has been completed.

Examples:

```
FOR I := 1 TO J DO LCNT :=I+J
FOR TINT := GREEN DOWNTO RED
     DO BEGIN
        LCNT := LCNT + 4;
        MEMBRS [LCNT].NAME [I] :=
             TITLE [J+I]
        END
```

3.8 WITH STATEMENT

The WITH statement provides a simplified notation for specifying a field identifier within an associated statement. Specifying a record variable in the WITH statement implies its existence before each of the field identifiers in the associated statement. The form of the WITH statement is:

WITH record-variable /, record-variable\ ... DO statement

There can be no assignment within the associated statement to any constituents of a record variable that appears in the WITH statement. The record variable(s) are considered fixed for the execution of the associated statement.

Example:

```
WITH MEMBRS [LCNT] DO
   BEGIN AGE := AGE +1;
     IF AGE > 16 ∧ ¬ HUMAN = ADULT
   THEN BEGIN HUMAN := ADULT;
              ASEX := MALE;
              FATHER := FALSE
        END
   END
```

The equivalent coded sequence not utilizing the WITH statement would appear,

```
MEMBRS [LCNT] .AGE := MEMBRS [LCNT] .AGE +1;
IF MEMBRS [LCNT] .AGE > 16 ∧
 ¬ MEMBRS [LCNT] .HUMAN = ADULT
THEN BEGIN MEMBRS [LCNT] .HUMAN := ADULT;
    MEMBRS [LCNT] .ASEX := MALE;
    MEMBRS [LCNT] .FATHER := FALSE
END
```

Note that if the scalar variable LCNT were modified during the sequence, the WITH statement could not be used.

3.9 PROCEDURE STATEMENT

A procedure statement serves to execute (or call) a procedure. The procedure statement may contain a list of actual parameters that replace the corresponding formal parameters of the procedure declaration. The general form for specifying a procedure statement is:

procedure-identifier /(actual-param /, actual-param \ . . .)\

Examples:

BININT

GETBNS (BP)

SINCOS (MULTI, RADIANS, SLANTRG)

3.10 EMPTY STATEMENT

The empty statement consists of no information at all. It may appear at any position where a statement is appropriate.

STANDARD PROCEDURES 4

A set of intrinsic procedures are associated with PASCAL. The user may assume that any of these is available to be called.

4.1 NEW

NEW is the dynamic allocation procedure. The PASCAL user will be provided storage for a variable with a NEW call. There are two calling sequences for the NEW procedure:

NEW (p)

NEW (p, t, /t\, . . .)

Where p is a pointer variable.

t is a tag field value.

The p pointer variable is bound to a variable type that allows the PASCAL compiler to determine the amount of storage to allocate. Upon return to the caller the p pointer contains the pointer value, which may be used in accessing the newly acquired variable.

The second form of the calling sequence may be used when accessing a record type variable that contains variants that may affect the size of the allocated space. The specification of the tag field value(s) provides the compiler with sufficient information to allocate the minimum storage that will accommodate the requested record structure.

4.2 PACK

PACK provides the mechanism for moving and transforming two characters in a CHAR-type array to a packed array. The PACK calling sequence is:

PACK (uparry, index, parry)

Where uparry is the source unpacked array of characters.

index is the character index into the unpacked array.

parry is the receiving packed array.

4.3 UNPACK

UNPACK provides the opposite capability as the PACK procedure. UNPACK moves and transforms two characters from a CHAR-type packed array to an unpacked array. The UNPACK calling sequence is:

UNPACK (parry, uparry, index)

Where parry is the source packed array of characters.

uparry is the receiving unpacked array.

index is the character index into the unpacked array.

4.4 APPEND

The APPEND procedure left-shifts (noncircular) an integer variable and then performs a logical OR of another integer into the shifted variable. The APPEND calling sequence is:

APPEND (var, shift, orval)

Where var is the variable that is to be left-shifted and will receive the result.

shift is the number of bit positions to left-shift.

orval is the value to be logically ORed into var.

After execution shift and orval are unchanged.

4.5 INSERT

The INSERT procedure left-shifts (noncircular) an integer value and then performs a logical OR of the shifted value into a receiving variable. The INSERT calling sequence is:

INSERT (orval, shift, var)

Where orval is the value to be left-shifted.

shift is the number of bit positions to left-shift.

var is the receiving variable into which the shifted value will be logically ORed.

After execution orval and shift are unchanged.

4.6 ADDR

The ADDR procedure stores the address of the first parameter in the second parameter. The first parameter may be a variable or a level 1 procedure. The ADDR calling sequence is:

```
ADDR ( i,j )
```

where the address of i is stored into variable j.

4.7 RETADR

The RETADR procedure stores the return address of the procedure in which it appears in the parameter. The parameter is an integer variable. The RETADR calling sequence is:

```
RETADR ( i )
```

where the return address is stored into integer variable i.

4.8 RETURN

The RETURN procedure stores the contents of the parameter in the word reserved for the procedure's return address. The parameter is an integer variable. The RETURN calling sequence is:

```
RETURN ( i )
```

where the contents of integer variable i are stored into the return address word.

4.9 LOCK

The LOCK procedure inhibits interrupts and increments the global interrupt flag by one. The LOCK calling sequence is:

```
LOCK
```

4.10 UNLOCK

The UNLOCK procedure inhibits interrupts, decrements the global interrupt flag by one, and enables interrupts if the global interrupt flag is zero (after being decremented). The UNLOCK calling sequence is:

```
UNLOCK
```

Note that if the global interrupt flag becomes negative, QDEBUG is called with an error code of 6.

4.11 IINT

The IINT procedure causes the compiler to generate an inhibit interrupts instruction [IIN O]. The IINT calling sequence is:

```
IINT
```

4.12 EINT

The EINT procedure causes the compiler to generate an enable interrupts instruction [EIN O]. The EINT calling sequence is:

```
EINT
```

4.13 STREGS

The STREGS procedure causes the compiler to generate a store registers instruction [SRG i]. The parameter is a variable. The STREGS calling sequence is:

```
STREGS ( i )
```

where variable i serves as the operand of the store registers instruction.

4.14 LDREGS

The LDREGS procedure causes the compiler to generate a load registers instruction [LRG i]. The parameter is a variable. The LDREGS calling sequence is:

```
LDREGS [i]
```

where variable i serves as the operand of the load registers instruction.

4.15 RESET

The RESET procedure replaces the contents of the pointer to the next available word in the dynamic variable area with the contents of the parameter. The parameter is a pointer variable. The RESET calling sequence is:

```
RESET ( i )
```

where i is a pointer variable.

4.16 INST

The INST procedure allows the user to specify instructions to be generated in-line. The INST procedure has a variable number of parameters [at least one]. The parameters may be constants, variables, or level 1 procedures. The generated instruction(s) reflects constants that will not be modified and address constants for variables and level 1 procedures. The INST calling sequence is:

```
INST (P1,P2, ..., Pn)
```

where $n \geq 1$ and Pi are constants, variables, or level 1 procedures.

FUNCTION PROCEDURES 5

PASCAL provides a specialized procedure known as a function procedure. It is declared in a form that is very similar to a procedure; however, its special effect is to return a single value that may be used as a variable in an expression of a PASCAL statement.

5.1 FUNCTION DECLARATION

A function procedure is declared in the local procedure declarations segment of a procedure or function procedure. The basic form of a function procedure is:

```
FUNCTION heading information
      D/D segment
      /local procedure declarations\
BEGIN
      action segment
END
```

The heading information provides the following:

- It indicates the start of the function procedure declaration.
- It assigns a function identifier for calling this function procedure.
- It defines the form of the calling sequence that must be presented with the function identifier when this function procedure is called.
- It declares the type of the returned value when this function procedure is called.

The general form of the function procedure heading is:

```
FUNCTION function-identifier /(formal-parameter-section)\ : type
```

The formal parameter section is defined in Section 2.2, Procedure Heading.

The D/D segment is specified in the same manner as a procedure.

The local procedure declaration segment may be used to declare procedures and/or function procedures.

The action segment of a function procedure is identical to that of a procedure, but it is necessary to specify at least one assignment statement of the form:

function-identifier := return-value

to establish the value that is to be returned to the caller.

A function has at least one parameter.

5.2 FUNCTION CALL

A function procedure is called to represent a computed value within a PASCAL statement. The call essentially stands in for a variable and may be specified at any position where a variable of corresponding type may be specified. The type of value returned by a function procedure is called the function type.

A sample recursive (calls itself) function procedure declaration is:

```
FUNCTION LEFTSHFT (VAL:INTEGER;CNT:INTEGER) : INTEGER;
BEGIN
   LEFTSHFT := VAL;
  IF CNT > 0 THEN LEFTSHFT := LEFTSHFT (VAL*2, CNT-1)
END
```

5.3 ABS

The ABS function procedure computes the absolute value of a presented integer.

ABS (x)

where x is the presented parameter of type integer.

5.4 SQR

The SQR function procedure computes the square (x^2) of a presented integer.

SQR (x)

where x is the presented parameter of type integer.

5.9 PRED

The PRED function procedure computes the predecessor value (the next lowest member in order) of a presented scalar value.

```
PRED (x)
```

where x is the presented parameter, which may be of any scalar type. The result is unpredictable if the presented parameter is the lowest member of the presented scalar type.

5.5 ODD

The ODD function procedure computes x MOD 2 for a presented integer.

```
ODD (x)
```

where x is the presented parameter of type integer.

5.6 ORD

The ORD function procedure computes the ordinal number (internal integer equivalent) of a presented character.

```
ORD (c)
```

where c is the presented parameter of type character. An integer-type result is returned.

5.7 CHR

The CHR function procedure generates the character from a presented ordinal number.

```
CHR (x)
```

where x is the presented parameter (an ordinal number of type integer). A char-type result is returned.

5.8 SUCC

The SUCC function procedure computes the successor value (the next highest member in order) of a presented scalar value.

```
SUCC (x)
```

where x is the presented parameter that may be of any scalar type. The result is unpredictable if the presented parameter is the highest member of the presented scalar type.

PASCAL PROGRAM 6

The main (or level 0) procedure defines a PASCAL program. It contains a complete set of local procedures and local function procedures. A PASCAL program has the form of a procedure declaration without a procedure heading. The final END keyword is followed by a period (.) character.

6.1 GLOBAL DATA

The variables that are declared in the D/D segment of the main procedure are considered global to the entire set of nested procedure and function procedure declarations. Under PASCAL, the level 0 variables will generate a standalone relocatable object deck with the assigned object deck name of GLOBL$.

Each of the global variable names that is represented in the GLOBL$ object text will be implicitly declared as externally defined so that separately assembled modules may make external references to them.

In addition, PASCAL includes, as externally defined, all constant definitions which have been defined in the CONST part and all field names that have been defined in the TYPE declaration part of the level 0 D/D segment. In the case of field names, the values assigned are the same as for an entry field definition:

1. The relative word position within the record to the named field
2. The start bit and bit length for the field

Within the context of the PASCAL language, it is syntactically correct to use the identical field name in two separate record definitions; however, a specification of this kind at level 0 will generate multiple external definitions.

Note that the PASCAL language does not permit an externally defined variable (one that is defined in a separate assembly/compilation) to be referenced by a PASCAL procedure.

6.2 EXTERNAL PROCEDURES/FUNCTIONS

Under PASCAL, each level 1 procedure or function procedure declaration generates a separate relocatable object text deck. The level 1 procedure identifier is defined as the entry point name by which any call from another level 1 procedure, the main procedure, or a separately assembled/compiled procedure may be resolved as a call to an external procedure.

In PASCAL all undefined procedure references are assumed to be calls to:

1. A level 1 procedure that has yet to be presented to the compiler (PASCAL is a single-pass compiler),
2. A separately compiled level 1 PASCAL procedure
3. A separately assembled subroutine (an honorary level 1 PASCAL procedure).

This implies that the generated code for the procedure calls to the undefined names includes an external reference declaration for that name.

Undefined function procedure references are considered to be errors; therefore, a function procedure call to:

1. A level 1 function procedure that has yet to be presented to the compiler
2. A separately compiled level 1 PASCAL function procedure
3. A separately assembled subroutine (an honorary level 1 PASCAL function procedure)

must be explicitly declared with a prestatement of its specification format (a forward declaration). The syntactical description of this prestatement is presented in Section 4.6, Forward Reference Declaration.

This capability allows arbitrary memory placement of all level 1 procedures via the link editing process. The main procedure's object text, less the global variables, will itself exist in a standalone object text deck with the compiler assigned deck name of MAIN$.

6.3 EXAMPLE PASCAL PROGRAM

The following example program is presented to show the relationship between the source of a PASCAL program and the entry externals generated in the associated object decks:

ENT — Entry definition

EXT — External reference

ENF — Entry field definition

EXF — External field reference

```
CONST
     BIGNUM = $7FFF;

TYPE
     RECA = PACKED RECORD
               GF1  :INTEGER;
               GF2  :BOOLEAN;
               GF3  :0..$7FFF
            END;
VAR
     X,Y  :INTEGER;
       Z  :RECA;

PROCEDURE  L1A;  ↱LEVEL 1↰
     VAR X1, Y1 :INTEGER;
     BEGIN
          X1 := 2; Y1 := 3;
          X := X1; Y := Y1;
          L1B
     END;

PROCEDURE  L1B;  ↱LEVEL 1↰
     VAR X2 :INTEGER;
          PROCEDURE L2B;↱LEVEL 2↰
          BEGIN X2 := X END;
     BEGIN
          L2B; L1A;
          Z.GF2 :=TRUE
     END;

BEGIN
     L1B
END.
```

```
Object Deck - GLOBL$
     ENF:  GF2
           GF3
     ENT:  BIGNUM
           GF1
           X
           Y
           Z
           GLOBL$

Object Deck - L1A

     ENT:  L1A
     EXT:  L1B

Object Deck - L1B

     ENT:  L1B
     EXT:  L1A

Object Deck - MAIN$
     ENT:  MAIN$
     EXT:  L1B
```

MSOS FEATURES 7

When a PASCAL program is executed in an MSOS (Mass Storage Operating System) environment, it may make use of MSOS input/output routines and monitor requests; also, to allow the program to execute in the MSOS protected area, an option to generate run-anywhere code is provided. Further information regarding these features is available in the MSOS and MS FORTRAN Version 3A/B reference manuals.

(Note that recursive procedures and the use of dynamic variables is allowed for PASCAL programs executing in the unprotected area but not for programs executing in the protected area.)

7.1 INPUT/OUTPUT

PREAD and PWRITE statements are used for both formatted and binary I/O. REWIND, BACKSPACE, and ENDFILE statements are used for manipulating magnetic tape files.

In the following:

lu is an integer constant or variable used to identify the logical unit.

Standard MSOS logical units are referenced as follows:

1 = input
2 = binary output
3 = list output
4 = comment

Other logical units are referenced using their actual assigned numbers.

a is a packed character array containing format information. The first character in "a" is a left parenthesis "("; a right parenthesis ")" follows the last format descriptor.

list is a series of variables separated by commas and terminated by a semicolon. Variables which occupy one or more full words may appear in the list.

7.1.1 PREAD — FORMATTED

PREAD(lu, a)list;

Data input from logical unit lu is scanned and converted according to the format information in array a, and then transferred to the variables in the list.

7.1.2 PREAD — BINARY

PREAD(lu)list;

Data input from logical unit lu is transferred without modification to the variables in the list.

7.1.3 PWRITE — FORMATTED

PWRITE(lu, a)list;

The values in the list are converted according to the format information in array a, and then output to logical unit lu.

7.1.4 PWRITE — BINARY

PWRITE(lu)list;

The values in the list are output without modification to logical unit lu.

7.1.5 REWIND

REWIND(lu);

The logical unit lu is positioned at its load-point.

7.1.6 BACKSPACE

BACKSPACE(lu);

The logical unit lu is positioned at the beginning of the preceding block.

7.1.7 ENDFILE

ENDFILE(lu);

An endfile record is written on logical unit lu.

7.2 FORMAT DESCRIPTORS

The format descriptors are: I, Z, A, R, H, *, and X.

In the following:

w, n are non-zero integer constants representing the width of the field in the external character string.

r indicates the repeat count is an optional non-zero integer constant indicating the number of times to repeat the succeeding descriptor.

Format descriptors are separated by commas and/or slashes; slashes indicate end-of-record.

7.2.1 I — INPUT (rIw)

The I descriptor is used to input decimal integer values. The input field consists of an integer subfield and may contain only the characters +, -, 0 through 9, or blank. When a sign appears, it must precede the first digit in the input field. Blanks are interpreted as zeros. The value is stored right-justified in the specified variable.

Examples: 2I6, I4, I5

7.2.2 I — OUTPUT (rIw)

The I descriptor is used to output decimal integer values. The output quantity occupies w output record positions right-justified in the field w. If the field w is larger than the number required, the output quantity is right-justified with blank fill on the left. If the field is too short, it is filled with asterisks.

Examples: 2I6, I7, I8

7.2.3 Z — INPUT (rZw)

The Z descriptor is used to input hexadecimal integer values. The input field w consists of a string of hexadecimal integer characters; blanks are interpreted as zeros.

Examples: Z4, Z3, Z2

7.2.4 Z — OUTPUT (rZw)

The Z descriptor is used to output hexadecimal integer values. The output quantity occupies w output record positions right-justified in the field w. It is an unsigned hexadecimal integer value, with a maximum absolute value of FFFF.

Examples: Z6, Z7, Z8

7.2.5 A — INPUT (rAw)

On input, the A descriptor accepts characters as list elements. If the field width w is two or more, the right-most two characters from the external input field are stored as the list element. If w equals one, the character from the external input field is left-justified in storage with a trailing blank.

Examples: 2A2, A1

7.2.6 A — OUTPUT (rAw)

The A descriptor outputs w characters from a two-character list element. If w is two or more, the two characters from memory appear right-justified in the external output field preceded by blanks. If w equals one, the left-most character from memory is stored in the output field.

Examples: 3A3, A1

7.2.7 R — INPUT (rRw)

On input, the R descriptor accepts characters as list elements. If the field width w is two or more, the right-most two characters from the external input field are stored as the list element. If w equals one, the character from the external input field is right-justified in storage with a leading hexadecimal 00.

Examples: 2R2, R1

7.2.8 R — OUTPUT (rRw)

The R descriptor outputs w characters from a two-character list element. If w is two or more, the two characters from memory appear right-justified in the external output field preceded by blanks. If w equals one, the right-most character from memory is stored in the output field.

Examples: 3R3, R1

7.2.9 H — OUTPUT (nH)

The H descriptor outputs characters, including blanks, in the form of comments, titles, and headings. n is an unsigned integer specifying the number of characters to the right of H that will be transmitted to the output record.

Examples: 6HABCDEF

7.2.10 * — OUTPUT (*ccc...*)

The literal descriptor (*) causes the string of characters between the *s to be transmitted to the output record.

Example: *THIS IS A COMMENT*

7.2.11 X — INPUT (nX)

The X descriptor causes n characters to be skipped on input.

Example: 5X

7.2.12 X — OUTPUT (nX)

The X descriptor causes n blanks to be inserted in the output record.

Example: 7X

7.3 PROTECTED AREA I/O CONSIDERATIONS

SETBFR must be called prior to performing formatted I/O in programs residing in the protected area.

Calling sequence: SETBFR (buffer, length)

Where: buffer is the starting location of the user's buffer.

length is the length of the user's buffer.

The first 18 words of the buffer contain the calling sequence for the I/O request and information for re-entrancy. The remainder contains the input/output data.

7.4 MONITOR REQUESTS

The following monitor requests may be made from a PASCAL program: READ, FREAD, WRITE, FWRITE, SCHEDL, TIMER, LINK, DISPAT, and RELESE.

7.4.1 READ, FREAD, WRITE, FWRITE

Calling sequence: name (lu, buffer, length, completion, flag, temp);

Where: name is READ, FREAD, WRITE, or FWRITE

lu is the mode and logical unit. The logical unit number is right-justified in the word and bit 12 indicates the mode (0 = binary mode, 1 = ASCII mode).

buffer is an area in memory where data is read into or written from.

length is the number of words to be read or written.

completion is the location to which control is returned after completion of the I/O operation.

flag is a packed word.

Bits

0 - 3	completion priority
4 - 7	request priority
8 - 15	0

temp is an eight-word area for building the calling sequence to the monitor.

Example:

```
          ADDR (100, COMPLT);
          FWRITE (LU, BUF, LENGTH, COMPLT, FLAG, TEMP);
    100   I := J;
          .
          .
          .
```

7.4.2 SCHEDL

Calling sequence: SCHEDL (p, flag, parameter, temp);

Where: p is the requested program to be scheduled at the completion priority specified by flag.

flag is a packed word with the completion priority in bits 0 through 3 and an indicator in bits 8 through 11. Indicator settings are:

0	p is a statement label
1	p is an index to the directory
2	p is an external core-resident main program

parameter is a positive integer which may be passed to the scheduled program. The scheduled program obtains the parameter by calling the integer function LINK.

temp is a four-word area in which the scheduler call is generated.

Example: SCHEDL (ABC, FLAG, 10, TEMP);

7.4.3 TIMER

Calling sequence: TIMER (p, flag, time, temp);

Where: p is the program to be given control at the completion priority specified by flag after the time interval specified by time has expired.

flag is a packed word containing the completion priority in bits 0 through 3, a unit of time code in bits 4 through 7, and an indicator in bits 8 through 11. Indicator settings are:

0-p is a statement label
1-p is an index to the directory
2-p is an external core-resident main program

Time code settings are:

0 system time units
1 1/10 second
2 1 second
3 1 minute

time is the time interval to delay before scheduling the program, p, at the completion priority specified by flag. At the end of the time interval, the core clock is passed to the requested program as a parameter. To obtain this parameter, the integer function LINK must be called.

temp is a four-word area in which the timer call is generated.

Example: TIMER (ABC, FLAG, 5, TEMP);

7.4.4 LINK

Calling sequence: LINK (0);

The function value is:

1. The passed parameter from a scheduler call if LINK is called at the start of the scheduled program.
2. The value of the core clock if LINK is called at the start of a program called by a TIMER request.

3. The error flag at the completion of I/O if LINK is called at the completion location.

Example: I : = LINK (0);

7.4.5 DISPAT

Calling sequence: DISPAT;

Control is given to the dispatcher in the monitor to start the next highest priority program.

7.4.6 RELESE

Calling sequence: RELESE;

All programs that have been allocated protected core must return memory to the core allocator when they are finished. This statement must be the last executed statement in the MAIN$ program. (Note that RELESE may only be called from a MAIN$ program and that the MAIN$ program should precede all procedures when program execution takes place in the protected area.)

7.5 RUN-ANYWHERE PROGRAMS

The run-anywhere option causes the compiler to replace program relocatable addressing with relative addressing. Because addresses are relative rather than relocatable, the loader does not have to modify a program after the program has been loaded. Run-anywhere programs may be executed in the MSOS protected area.

The RELATIVE declaration is used to specify those procedures which are to be referenced with relative rather than absolute addressing. It precedes the LABEL declaration in the global definitions.

Example:

```
$M+, Y+
RELATIVE
    pgm1, pgm2, pgm3;
LABEL
    10, 20;
CONST
    K1 = 1,
    K2 = 2;
TYPE
    T1 = 0..3;
.
.
.
```

SYNTACTICAL DESCRIPTIONS OF PASCAL ELEMENTS A

Reference Section

Procedure Structures 2.1

```
PROCEDURE    proc-identifier /(formal-parameter-list)\
    /LABEL   statement-label /, statement-label\...;\
    /CONST   constant-definition; /constant-definition;\...\
    /TYPE    type-definition; /type-definition;\...\
    /VAR     variable-declaration; /variable-declaration;\...\
    /VALUE   value-assignment; /value-assignment;\...\
BEGIN
     action segment
END
```

Formal Parameter List 2.2

```
/VAR      variable-parameter/, variable-parameter\...:type-identifier\
/value-parameter/, value-parameter\...:type-identifier\
```

Statement Label 2.4.3

```
unsigned-integer
```

Constant Definition 2.3.2

```
identifier = constant-value
```

Type Definition 2.3.3

```
type-identifier = data-type-definition
```

Variable Declaration 2.3.4

```
variable-identifier/, variable-identifier\...:data-type-definition
variable-identifier/, variable-identifier\...:type-identifier
```

	Reference Section
Value Assignment	2.3.5
variable-identifier = variable-initialization	
Data Type Definitions	
Scalar:	2.3.3.1
type-identifier = (constant-identifier/,constant-identifier\...)	
Subrange:	2.3.3.1
type-identifier = minimum-constant .. maximum-constant	
Array:	2.3.3.2
type-identifier = ARRAY [index-type/,index-type\...] OF component-type	
Record:	2.3.3.2
type-identifier = RECORD /fixed-part\ /variant-part\ END	
Fixed Part:	
field-identifier /,field-identifier\...:type /;field-identifier/,field-identifier\...:type\...	
Variant Part:	
CASE tag-field: tag-type-identifier OF case-label/,case-label\...:(/fixed-part\/variant-part\) /;case-label/,case-label\...:(/fixed-part\/variant-part)\...	
Set:	2.3.3.2
type-identifier = SET OF scalar-type	
Pointer:	2.3.3.3
type-identifier = ↑defined-type-identifier	

	Reference Section
Simple Action Statements	
Assignment:	3.4.3.1
receiving-variable :=expression	
destination-variable :=source-variable	
set-variable :=[/constant/,constant\...\]	
function-identifier :=return-value	
pointer-variable :=pointer-variable ± arithmetic-expression	
GOTO:	3.4.3.2
GOTO statement-label	
GOTO EXIT statement-label	
IF:	3.4.3.3
IF boolean-expression THEN statement /ELSE statement	
CASE:	3.4.3.4
CASE expression OF	
constant/,constant\...:statement	
/;constant/,constant\...:statement ...	
END	
WHILE:	3.4.3.5
WHILE boolean-expression DO statement	
REPEAT-UNTIL:	3.4.3.6
REPEAT statement/;statement\... UNTIL boolean-expression	
FOR:	3.4.3.7
FOR control-variable :=initial-expression TO final-expression DO statement	
FOR control-variable :=initial-expression DOWNTO final-expression DO statement	

	Reference Section
WITH:	3.4.3.8
WITH record-variable/,record-variable\... DO statement	
Procedure Call:	3.4.3.9
procedure-identifier/(actual-param/,actual-param\...)	
Forward Reference Declaration	2.5
PROCEDURE proc-identifier /(formal-parameters)\ ;FORWARD	
FUNCTION function-identifier /(formal-parameters)\ :type;FORWARD	

PASCAL COMPILER OPTIONS B

Compiler options are presented to the PASCAL compiler in two forms:

1. As parameters passed in the job control statement call of the PASCAL compiler
2. As option comments that appear in the source statements presented to the PASCAL compiler

PASCAL Call Options

1. P = lfn

   lfn is the logical file name on which the PASCAL source program resides. (Default: P = INPUT)

2. L = lfn

   lfn is the logical file name onto which the PASCAL compiler writes the source listing. (Default: L = OUTPUT)

3. O

   O indicates that a listing of the object code is to be written. (Default: no object code listing)

4. CSET = ch

   ch is the name of the character set used by the operating system. ch is 63 for the CDC 63-character set, and ch is 64 for the CDC 64-character set. (Default: CSET=63)

Example PASCAL call:

```
PASCAL(P=COMPILE, O)
```

Option Comments

PASCAL option comments control the mode of generated code or provide listing control on the generated output listing. They may be inserted at any position in the program. Each of the available options is designated by a single letter code immediately followed by a plus sign (+), which turns the option on, or a minus sign, which turns the option off. More than one option may be listed in an option comment by separating each specification with a comma.

An option comment takes the general form of a comment (i.e., it is surrounded by ⌐→ and ↓ where the first character following the ⌐→ is a dollar sign ($)). The option codes and their meanings are:

Code	Meaning
A	For each assignment to a subrange variable, check whether the value assigned lies within the specified subrange. (Default: A-)
B	For the next external procedure call, generate an SJQ (rather than RTJ) instruction. This option is in effect for one call only. (Default: B-)
C	Reserved for compiler maintenance.
D	For each division operation, compile instructions that will check for a zero divisor. (Default: D-)
E	For each specification of a record variable containing a variant case, do not generate space for the tag field. (Default: E-)
G	For all following procedures and function procedures, compile instructions that will protect from interrupting the stack management instructions that appear in the entry/exit code. (Default: G-)
H	Output a heading with page number for each page output. (Default: H-)
I	For all following procedures and function procedures, compile an interrupt lock-out upon entry and an interrupt unlock at exit. (Default: I-)
J	Page eject before printing the next line. (Default: J-)
	Allow for execution of object programs under the MSOS operating system. (Default: M-)
N	For the following level 0 variable, field, and constant symbol names, suppress the generation of external definitions within the object text. (Default: N-)
O	For each call of a recursive procedure or recursive function procedure, compile instructions that will check (at runtime) for stack overflow. (Default: O-)
R	For all following procedures and function procedures, compile instructions that allow for recursion. (Default: R+)
S	Suppress the source listing following the specification of this option. (Default: S-)
T	Set options A, D, O, V, and X (on or off). (Default: options set individually)
V	For each request for space from the dynamic variable area (via NEW), compile instructions that will check for area overflow. (Default: V-)
X	For each array variable specification, compile instructions that will check the specified array indices to determine if they lie within the array bounds. (Default: X-)
Y	Generate run-anywhere object code. (Default: Y-)

Example option comment:

⌐→$O+, X-, D-, E- ↓

PASCAL COMPILATION ERROR MESSAGES C

Compilation errors are indicated by flagging the detected error.

The listing line following the source line in error begins with **** in the location field. At, or immediately after, the column in which the error was identified, an ↑ will be printed, followed immediately by a numeric code. The following table lists the codes and their meaning.

Error Code	Meaning
1	Scalar type expected
2	Integer too large
3	Error in constant
4	= expected
5	Field name declared twice
6	Bad range
7	Tag field type bad
8	Name declared twice
9	) expected
10	: expected
11	Identifier expected
12	Identifier not declared
13	Index must be of scalar type
14	0F expected
15	Ten or more errors on this line
16	Procedure declared twice
17	END expected
18	Error in type declaration
19	Range restricted to 0 through 15
20	Error in VALUE part
21	Too many arguments for this procedure
22	Value is out of range

Error Code	Meaning
23	Too many relative procedure names or exit labels
24	Error in declaration part
25	Lowbound is greater than highbound
26	Not a variable identifier
27	Label too large
28	Symbolic subrange type not allowed
29	Parameter missing in function declaration
30	Too many unique external type references
31	Too many unique external references
32	Variable or field identifier expected
33	Expression too complicated
34	Type of variable should be array
35	Type of expression must be scalar
36	Conflict of index type with declaration
37	] expected
38	Type of variable should be record
39	No such field in this record
40	Type of variable should be pointer
41	Field name expected
42	Illegal symbol in expression
43	Undefined label
44	Illegal type of parameter in standard function or procedure
45	Type identifier in statement part
46	Procedure used as function
47	Type of standard function parameter should be integer
48	Index out of range
49	[expected
50	Illegal type of operand
51	v cannot be used as monadic operator
52	:= expected
53	Assignment not allowed

Error Code	Meaning
54	Illegal symbol in statement
55	Type or constant identifier
56	THEN expected
57	Type of expression is not Boolean
58	; expected
59	DO expected
60	Illegal parameter substitution
61	Label expected
62	Illegal type of expression
63	Constant expected
64	Type declared twice
65	Bad function type
66	Tag field missing for this variant
67	UNTIL expected
68	Only = and ≠ allowed here
69	Loop control variable must be simple and local or global
70	TO or DOWNTO expected
71	Too many cases in CASE statement
72	Number of parameters does not agree with declaration
73	Mixed types
74	Too many labels in this procedure
75	Too many constants, yet-undefined labels, or temporary storage references
76	Depth of procedure nesting too large
77	Label defined more than once
78	Too many exit labels
79	(expected
80	, expected
81	Too many exit labels or forward procedures
82	Too many nested WITH statements
83	Value declaration in recursive procedure
84	Too many constants in this procedure

Error Code	Meaning
85	Assignment to function identifier must occur in function itself
86	Actual parameter must be a variable
87	Packed field not allowed here
88	Operators < and > are not defined for powersets
89	Redundant operation on powersets
90	Procedure too long
91	Begin comment character (↦) imbedded in comment

PASCAL CORRELATION TABLE

CHAR	HOLLERITH	CHAR	HOLLERITH	CHAR	HOLLERITH
Blank	No punch	6	6	K	11-2
∨	11-0	7	7	L	11-3
≠	8-4	8	8	M	11-4
≡	0-8-6	9	9	N	11-5
$	11-8-3	:	8-2	O	11-6
Reserved	—	;	12-8-7	P	11-7
∧	0-8-7	<	12-0	Q	11-8
↑	11-8-5	=	8-3	R	11-9
(	0-8-4	>	11-8-7	S	0-2
)	12-8-4	↓	11-8-6	T	0-3
*	11-8-4	≤	8-5	U	0-4
+	12	A	12-1	V	0-5
,	0-8-3	B	12-2	W	0-6
-	11	C	12-3	X	0-7
.	12-8-3	D	12-4	Y	0-8
/	0-1	E	12-5	Z	0-9
0	0	F	12-6	[	8-7
1	1	G	12-7	≥	12-8-5
2	2	H	12-8	]	0-8-2
3	3	I	12-9	¬	12-8-6 *
4	4	J	11-1	↑	0-8-5
5	5				

*When using the 200 Users Terminal, the % [8-6] character should be input in place of the ¬ [12-8-6] character.

COMPILER LIMITS

E

Description	Maximum Number
Active exit labels	10
Active FORWARD declarations	10
Parameters for a procedure	20
Forward-referenced types used in pointer type definitions	21
External or forward-referenced procedures referenced in a level 1 procedure	40
Constants local to a level 1 procedure and its subprocedures	100
Labels for a procedure	100
CASE statement labels	100
Relative procedures	100
Words written to the binary file for a level 1 procedure	15360
Local symbols	1760
Global (level 0) symbols	1536

These limits can be changed. For further information see the NOS Installation Handbook and the NOS/BE Installation Handbook listed in the preface.

FORMAT PROGRAM F

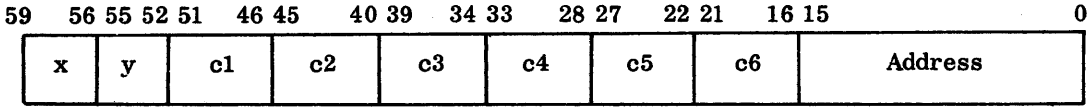
The format program converts the intermediate binary output from the PASCAL compiler into a form that is compatible with the MSOS relocatable loader and is acceptable as input to the link editor and the library maintenance program. The format program requires the file it reads to have the logical file name PASCLGO.

INPUT

Eight types of data are input to the format program:

Format 1 data	NAM (0010)	Name data
	ENT (1000)	Entry point data
	ENF (0000)	Entry field data
	EXT (1010)	External name data
	EXF (1110)	External field data
	XFR (1100)	Transfer address data
Format 2 data	RBD (0100)	Command sequence data
	BZS (0110)	Zero storage data

Format 1 data has the following form:

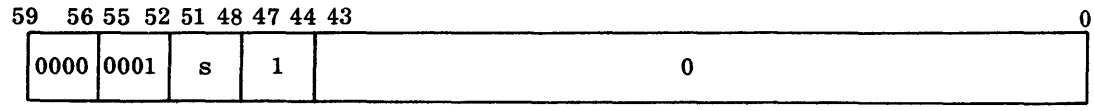


Where: x = 0010, NAM
1100, XFR
1000, ENT
0000, ENF
1010, EXT
1110, EXF

y = 0000, relocatable ENT
relocatable EXT

y = 0001, absolute ENT
relative EXT

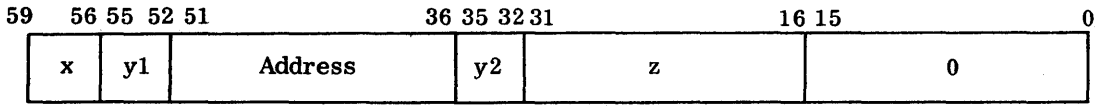
ENF has an additional word:



Where: s = start bit

l = length

Format 2 data has the following form:



Where: x = 0100, RBD
0110, BZS

y1 = 0001, relocatable
0011, global

y2 = 0000, absolute
0001, relocatable
0011, global

z = the RBD instruction or the BZS size

Note: For BZS, y2 is always zero.

OUTPUT

Output is as defined in Chapter 12 of the MSOS reference manual. As with the binary card format, the various blocks (i.e., NAM, EXT, RBD, etc.) are preceded by a sequence number word and a complemented word-count word, and are followed by a checksum word.

NOTES

1. Global variables are classed as labeled COMMON and their object deck is GLOBL$.
2. ENT items are grouped to form ENT blocks.
3. EXT items are grouped to form EXT blocks.
4. The Format program locates the ENT and EXT blocks after the last RBD block.
5. There is only one NAM block per object program, and it is the first block of the object program.
6. There is only one XFR block per object program, and it is the last block of the object program.
7. External references (EXT) are references to CYBER 18 PASCAL level 1 procedures, assembly language subroutines, and main program entry points.
8. Relocatable entry points (ENT) are names of level 1 procedures, global variables, and main program entry points.
9. Absolute entry points (ENT) are for fields within records defined in the global section.
10. Main programs are given the name MAIN$.

ERROR MESSAGES

SIZE OF ARRAY ISAVE EXCEEDED---TERMINATE PROCESSING

More than the maximum number of ENTs, ENFs, and EXTs are present in the object program being reformatted.

ILLEGAL INPUT DATA TYPE IN WORD xxx...xxx

There is bad data in the PASCLGO file.

STOP 7777 (in dayfile)

I/O error occurred while attempting to write on the LGO file.

The PASCAL cross-reference program is used to obtain cross-reference listings of programs written in PASCAL. The cross-reference program is written in 6000 PASCAL. The cross-reference is produced from the output listing produced by CYBER Cross PASCAL.

Example job set-up:

```
job card
ATTACH(PASCAL,ID=SCDD)  <-------  Attach the MP17 PASCAL.
ATTACH(PASXREF,ID=SCDD) <-------  Attach the XREF binary.
RFL(77000)
PASCAL(L=TEMP)          <-------  Compile the PASCAL program.
RETURN(PASCAL)
REWIND(TEMP)
ATTACH(PASCAL,PASBNO1,ID=SCDD) <-------  Attach the 6000 PASCAL.
RFL(125000)
PASCAL(LOAD=PASXREF,D=TEMP) <-------  Run XREF using MP17 output as the input.
7/8/9
... PASCAL source program ...
6/7/8/9
```

GLOSSARY

Action segment	The portion of a procedure that defines the logic by which the variables will be affected.
Constant	A literal representation of a fixed value that is associated with some data type; may be specified as a decimal, hexadecimal, or octal integer or as a character or a character string.
Declaration/definition segment	The portion of a procedure that describes the meaning of the various identifiers used in a program.
Expression	A construct that directs the execution of a computation.
Formal parameter	An identifier that represents the actual parameter to be substituted within the procedure program segment when it is to be executed.
Global procedure	A procedure that contains a nested, or local, procedure.
Identifier	A contiguous sequence of letters and decimal digits, beginning with a letter, that represents a constant, type definition, variable, procedure, or function to the PASCAL compiler.
Keyword	A reserved identifier that directs the compilation process.
Local procedure	A procedure that is nested within another procedure.
PASCAL	A high-level, algorithmic-type language patterned after ALGOL 60.
Procedure	The primary unit of program structure in PASCAL; the algorithm intended for execution on the processor. Analogous to a subroutine.
Value parameter	A data element whose value may be used by the procedure during processing.
Variable parameter	A data element for which the procedure may produce a result.

MSOS CONSIDERATIONS H

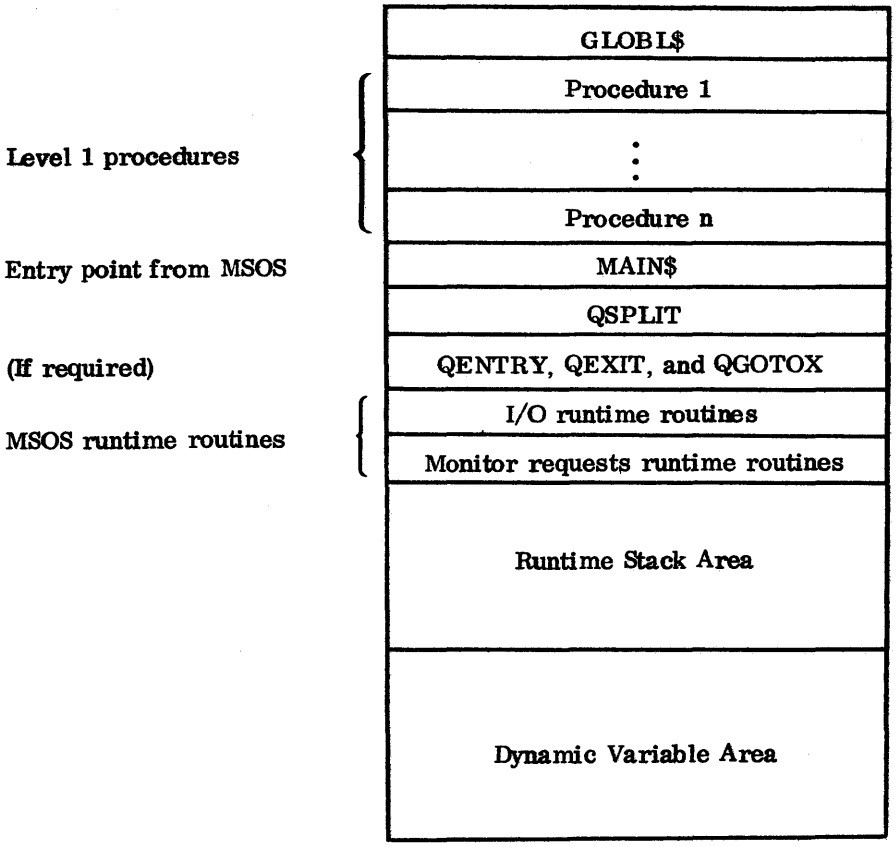
Executing PASCAL programs in the MSOS background (unprotected allocatable memory):

1. The MSOS option should be used when compiling PASCAL programs to run in the MSOS background.
2. The global variables as a whole are implemented as labeled COMMON, and they make up the object deck GLOBL$. GLOBL$ should be the first object program loaded as any of the level 1 procedures or the main program may reference global variables.
3. The main program (MAIN$) calls QSPLIT which performs the runtime stack and dynamic variable area initialization. To ensure that the MSOS relocatable loader transfers control to the main program, the object deck, MAIN$, should be the last object deck loaded (except for runtime routines).
4. Runtime routine QSPLIT divides the unused background space equally between the runtime stack and the dynamic variable area.
5. QSPLIT uses a table of externals to initialize the variables in the other runtime routines (QENTRY, QEXIT, and QGOTOX) which point to the top of the runtime stack. QENTRY, QEXIT, and QGOTOX need not be loaded if recursive procedures are not used.
6. If recursive procedures are used, the I register must be saved and restored when performing I/O and making monitor requests.

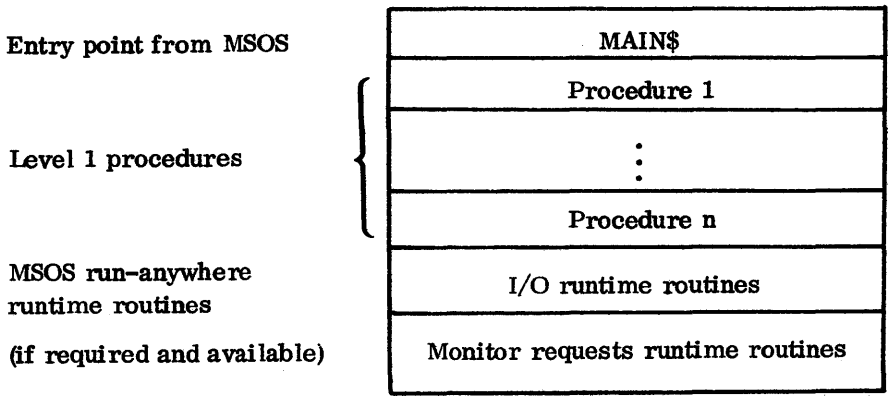
Executing PASCAL programs in the MSOS protected area (protected allocatable memory):

1. The MSOS option and the run-anywhere option should be used when compiling programs to run in the MSOS protected area.
2. Because global variables are implemented as labeled COMMON; unless they are included at the time the system is built, PASCAL programs cannot use global variables. Therefore, the main program cannot access any variables, and as a result the bulk of the program must be implemented via level 1 procedures.
3. In general, the object deck GLOBL$ should be discarded and the object deck MAIN$ should be the first object program loaded for PASCAL programs executing in the MSOS protected area.
4. In general, the main program will call one or more level 1 procedures which will perform the required program functions. When processing is complete, control should be returned to the main program which then calls RELESE.

   Note that for protected programs the last instruction executed should be a call of RELESE; for PASCAL programs, that call can only successfully be made from a main program.
5. The program may perform I/O and monitor requests if the runtime routines are included at the time the system is built or if run-anywhere versions of the runtime are available.
6. Note that neither the runtime stack or the dynamic variable area are available for PASCAL programs executing in the MSOS protected area.



Layout of PASCAL Program in the MSOS Background Area



Layout of PASCAL Program in the MSOS Protected Area

INDEX