LEVEL II

RADC-TR-81-143
Final Technical Report
June 1981

AD A101061

COMPUTER PROGRAMMING MANUAL FOR THE JOVIAL (J73) LANGUAGE

Softech, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-143 has been reviewed and is approved for publication.

APPROVED: Donald L. Mark

DONALD L. MARK
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-81-143
2. GOVT ACCESSION NO.: AD-A 101 061
3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): COMPUTER PROGRAMMING MANUAL FOR THE JOVIAL (J73) LANGUAGE
5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. Sep 78 - Jan 80
6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s)
8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0188
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Softech, Inc. 460 Totten Pond Road Waltham MA 02154
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F 25320203
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS) Griffiss AFB NY 13441
12. REPORT DATE: June 1981
13. NUMBER OF PAGES: 390
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same
15. SECURITY CLASS. (of this report): UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A
16. DISTRIBUTION STATEMENT (of this Report): Approved for public release; distribution unlimited.
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report): Same
18. SUPPLEMENTARY NOTES: RADC Project Engineer: Donald L. Mark (ISIS)

Accession For	
NTIS GRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
JOVIAL (J73)
MIL-STD-1589A
Programming Manual
Higher Order Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This manual is a combined tutorial and reference manual for the JOVIAL (J73) language as defined in MIL-STD-1589A dated 15 March 1979. The main body of the manual describes the entire language, giving motivation and examples for each feature. This manual is intended for a reader who has had previous experience with assembly language or some higher order language. It does not teach the fundamentals of programming. On the other hand, the presentation is informal and non-

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

405 [illegible] JOB

mathematical.

CONTENTS

Chapter 1

INTRODUCTION

JOVIAL (J73) is a higher-order programming language. It is being implemented on many computer systems and used in many applications areas. Typical applications areas are avionics, command and control, and missile flight control.

Sufficient capability has been provided to permit programming of most command and control applications in JOVIAL (J73). It is intended that assembly language programs be combined with programs written in JOVIAL (J73) to form a total application software package. The assembly language programs can provide certain utility operations as well as all hardware-dependent activities such as input, output, and interrupt services.

The language independently processes procedures and functions of the units of an application. Standard subroutine linkage and argument transmission with a powerful compool file can be used to effectively modularize programs and control interfaces.

Permissable data structures are simple items, structured tables of simple items, and composite data blocks containing simple items and tables.

Types of data in data structures can be signed or unsigned integers; enumeration values, floating point numbers, fixed point (fractional) numbers, character strings, bits strings (logical), and pointers (address of data objects).

A full complement of language constructs permits looping, branching, conditional execution, procedure or function calls, and assignment of values to data elements.

1.1 THE PRINCIPAL FEATURES OF JOVIAL

The following paragraphs provide an introduction to the principal features of JOVIAL. They discuss values, storage, calculations, operators, built-in functions, flow of control, subroutines, programs, compiler directives, compiler macros, and, finally, the advanced features of the language.

1.1.1 Values

The kinds of values provided by JOVIAL reflect the applications of the language; they are oriented toward engineering and control programming rather than, for example, commercial and business programming. The JOVIAL values are:

1. Integer values, which are signed or unsigned whole numbers. They are used for counting. For example, an integer can be used to count the number of times a loop is repeated or the number of checks performed on a process.

2. Floating values, which are numbers with "floating" scale factors. They are used for physical quantities, especially when the range of measurement cannot be accurately predicted. For example, floating values are frequently used to represent distance, speed, temperature, time, and so on.

3. Fixed values, which are numbers with constant scale factors. They are sometimes used for physical quantities (primarily to save time and/or storage) when the range of the value is narrow and predictable. For example, fixed values might be used in a computation that had to run on a computer for which floating-point hardware was not available or was too slow.

4. Bit-string values, which are sequences of binary digits (bits). They are used for communication with "on-off" devices or to control parts of the program itself. For example, a bit-string could be used to represent settings of switches on a control console.

5. Character-string values, which are sequences of characters. They are used for communication with people. For example, a character-string could be sent to an operator terminal to report failure of a portion of the system.

6. Status values, which are special words. They are used to describe the status of the system, or a particular part of the system, at any given time. For example, status values of "V(OK)", "V(WEAK)", or "V(BAD)" can be used to indicate the condition of a power cell.

7. Pointer values, which are data addresses, meaningful only within the program. They are used to locate data indirectly. For example, a list of items can use pointers to connect each item to the next item in the list.

8. Table values, which are collections of values gathered together to form a single data object. They are used for the constructs called "arrays" and "structures" in other languages. For example, a table can be used to store temperature readings taken every 1Ø seconds during a given test period.

9. Block values, which are collections of values gathered into one region of memory. They are used to support memory management. For example, certain data that must be paged in and out of memory together can be placed in a block.

1.1.2 Storage

When a JOVIAL program is executed, each value it operates on is stored in an item. The item has a name, which is declared and then used in the program when the value of the item is fetched or modified.

An item is declared by a JOVIAL statement called a declaration statement. The declaration provides the compiler with the information it needs to allocate and access the storage for the item. Here is a statement that declares an integer item:

```
ITEM COUNT U 1Ø;
```

This declaration says that the value of COUNT is an integer that is stored without a sign in ten or more bits. The notation is compact: "U" means it is an unsigned integer, "1Ø" means it requires at least 1Ø bits. We say "at least" ten bits because the JOVIAL compiler may allocate more than ten bits. (That alloction wastes a little data space, but can result in faster, more compact code.)

JOVIAL does not require that you give the number of bits in the declaration of an integer item. If you omit it, JOVIAL supplies a default value that depends on which implementation of JOVIAL you are using. An example is:,

```
ITEM TIME S;
```

This statement declares TIME to be the name of an integer variable item that is signed and has the default number of bits. On one implementation of JOVIAL, this would be equivalent to the declaration:

```
ITEM TIME S 15;
```

The item TIME occupies 16 bits (including the sign). On another implementation, it would be equivalent to:

```
ITEM TIME S 31;
```

This and other defaults are defined in the user's manual for the implementation of JOVIAL you are using.

In this brief introduction, we cannot consider each kind of item in detail (as we just did for integer items). Instead, a list of examples follow, one declaration for each kind of value.

ITEM SIGNAL S 2;	A signed integer item, which occupies at least three bits and accomodates values from -3 to +3.
ITEM SPEED F 30;	A floating item, whose value is stored as a variable coefficient (mantissa) and variable scale factor (exponent). The "30" specifies thirty bits for the mantissa and thus determines the accuracy of the value. The number of bits in the exponent is specified by the implementation, not the program. It is always sufficient to accommodate a wide range of numbers.
ITEM ANGLE A 2,13;	A fixed item, whose value is stored with fixed scaling, namely two bits to the left of the binary point and thirteen fractional bits. Thus it accomodates a value in the range -4 < value < +4 to a precision of 1/(2**14).

ITEM CONTROLS B 1Ø; — A bit-string item, whose value is a sequence of ten bits. Thus it can accommodate, for example, the settings of ten on/off console switches.

ITEM MESSAGE C 8Ø; — A character-string item, whose value is a sequence of eighty characters. Thus it can accommodate, for example, the message "WARNING: Cooling system failure" (with plenty of character positions left over).

ITEM INDICATOR STATUS (V(RED),V(YELLOW),V(GREEN)); — A status item, whose value can be thought of as "V(RED)", "V(YELLOW)", or "V(GREEN)" but which is, in fact, compactly stored as an integer. Thus a programmer can assign "V(RED)" to a variable to indicate cooling system failure instead of using a (presumably non-mnemonic) integer.

ITEM HEAD P DATA; — A pointer item, whose value is the address of a data object of type DATA.

Items are just the scalar (single-value) data of JOVIAL. JOVIAL also has tables and blocks, which provide for arrays and other data structures.

An example of a table declaration is:

```
TABLE GRID (1:1Ø, 1:1Ø);
   BEGIN
   ITEM XCOORD U;
   ITEM YCOORD U;
   END
```

The table GRID has two dimensions. Each dimension contains ten entries. Each entry consists of two items, XCOORD and YCOORD.

An example of a block declaration is:

```
BLOCK GROUP;
   BEGIN
   ITEM FLAG B;
   TABLE DATA(1ØØ);
      ITEM POINT U;
   END
```

The block GROUP contains the item FLAG and the table DATA.

Items, tables, and blocks can also be declared using type-names. A type-name is defined in a type declaration. An example of a type declaration is:

```
TYPE COUNTER U 10;
```

The type-name COUNTER can be used to declare ten-bit integers. For example:

```
ITEM CLOCK COUNTER;
```

1.1.3 Calculations

In the simplest case, calculation is performed by an assignment statement. An example is:

```
AVERAGE = (X1 + X2)/2;
```

The right-hand-side of this assignment is a formula; it forms the sum of X1 and X2 and divides it by 2. The details of the operation depend on how X1 and X2 are declared. If X1 and X2 are declared float, the calculation is very likely to produce the expected result. In contrast, if the X1 and X2 are declared fixed, the scaling must be worked out by the programmer to make sure the calculation will succeed. And if X1 and X2 are declared character-string, the compiler will reject it because JOVIAL does not automatically convert values into the types required by operators.

In the example just given, the parentheses show that the addition is performed before the division. When parentheses are not given, JOVIAL recognizes the usual order of evaluation. Here is an example:

```
POLY = BETA*X1**2 - GAMMA*X2 + DELTA;
```

JOVIAL applies its "rules of precedence" to the formula in this assignment and thus interprets it as:

```
POLY = (((BETA*(X1**2)) - (GAMMA*X2)) + DELTA);.
```

The complete precedence rules are given in Chapter 11.

The examples just given illustrate the use of formulas on the right-hand side of an assignment statement. A formula can also appear as part of the left-hand side of an assignment statement; for example, as the subscript of an array. In addition to their important role in assignment statements, formulas can appear in many other places in the language: as actual parameters of functions and procedures, as the condition in an if-statement, and so on.

Since JOVIAL has quite a few kinds of values, it must have many ways of converting one kind of value into another kind. In most cases, you must explicitly indicate the conversion. An example is:

```
ITEM MARK U 10;
ITEM TIME F;
...
MARK = (* U 10 *) (TIME);
```

The value of the floating item TIME is converted to a ten-bit integer value before it is assigned to the ten-bit integer item MARK. If you leave the conversion operator out of this assignment, the compiler will report an error. The compiler catches situations in which one type of value is unintentionally assigned to or combined with a different type of variable.

1.1.4 Operators

The operations provided in JOVIAL reflect the applications of the language; they determine what the language can and cannot do. Thus JOVIAL is strong in numerical calculation and control logic, but has minimal operations for text processing.

JOVIAL does not have any operations for input-output or file maintenance because it is assumed that a JOVIAL program runs in a relatively specialized environment that provides subroutines for those operations.

Some of the operations of JOVIAL are represented by operators, others by built-in functions.

The JOVIAL operators are summarized in the following table:

Type	Operators	Operation
Arithmetic	+ -	prefix signs
	**	exponentiate
	* / MOD	multiply, divide, and modulus
	+ -	infix add and subtract
Relational	< > =	less than, greater than, equal
	<= >= <>	less than or equal, greater or equal, not equal
Logical	NOT	(prefix) "not"
	AND OR	"and", "or"
	XOR EQV	"exclusive or", "equivalent"

An arithmetic operator takes integer, float, or fixed values as its operands and produces an integer, float, fixed value as its result. Type classes cannot be mixed. For example, a fixed value cannot be added to a float value unless one is explicitly converted to the type of the other.

A relational operator compares any two values of the same type and produces a Boolean value as its result. A logical operator takes bit-string values and also produces a Boolean result. (A Boolean value is a one-bit bit-string, representing "true" or "false", depending on whether it is one or zero.)

The JOVIAL operators are described in detail in Chapter 11, where, for example, you will find the rules for operations on fixed values and for the comparison of such objects as character-strings and pointers.

1.1.5 Built-In Functions

The JOVIAL built-in functions provide advanced, specialized operations that are not covered by the JOVIAL operators. They are summarized in the following table:

Function	Result
LOC(r)	A pointer to the object referenced by r
NEXT(p,i)	A pointer to the i'th data object after the one selected by p
NEXT(s,i)	The i'th status value after status value s
BIT(b,i,n)	A string of n bits starting at the i'th bit of the bit string b
BYTE(c,i,n)	A string of n characters starting at the i'th character of the character string c
SHIFTL(b,n)	Bit string b shifted left by n bits
SHIFTR(b,n)	Bit string b shifted right by n bits
ABS(x)	Absolute value of x
SGN(x)	+1, Ø, or -1 for x>Ø, x=Ø, x<Ø
BITSIZE(x)	Logical size of x in bits
BYTESIZE(x)	Logical size of x in bytes
WORDSIZE(x)	Logical size of x in words
LBOUND(t,d)	Lower bound of d'th dimension of the table t
UBOUND(t,d)	Upper bound of d'th dimension of the table t
NWSDEN(t)	Number of bytes allocated to each entry of the table t
FIRST(s)	First status value in status list for s
LAST(s)	Last status value in status list for s

An example of the use of a built-in function is:

```
C = BYTE("ABCDEF",2,3);
```

The built-in function extracts "BCD" from the string "ABCDEF".

Two of the built-in functions, BIT and BYTE, can be used as pseudo-variables. In that form, they appear as the target of an assignment, and are interpreted "backwards". An example is:

```
C = "ABCDEF";
BYTE(C,2,3) = "XYZ";
```

This assignment changes the second, third, and fourth characters of C to "XYZ". The value of C after the assigment is therefore "AXYZEF".

1.1.6 Flow of Control

For structured flow of control, JOVIAL has an if-statement, a case-statement, and a loop-statement with an optional exit-statement. Examples of these statements follow.

Here is an example of an if-statement:

```
IF SPEED < LIMIT;
   FLAG = TRUE;
ELSE
   BEGIN
   FLAG = FALSE;
   VIOLATION = VIOLATION+1;
   END
```

If SPEED is less than LIMIT, this statement sets FLAG to TRUE and does nothing else. If SPEED is not less than LIMIT, the statement sets FLAG to FALSE and increments VIOLATION. The last four lines of the example are a compound statement; the BEGIN-END pair groups the assignments to FLAG and VIOLATION into a single compound statement controlled by the ELSE clause.

The ELSE clause of an if-statement is optional; when it is omitted, no action is taken when the condition is false. Furthermore, if-statements can be nested, so complicted control structures can be built up. When if-statements get large and complicated, however, you can sometimes use a case-statement to clear things up.

Here is an example of a case-statement:

```
CASE NUM;
   BEGIN
   (DEFAULT):                       TYPE=V(OUT'OF'RANGE);
   (1,3,5,7,11,13,17,19):           TYPE=V(PRIME);
   (2,4,6,8:10,12,14:16,18,20):     TYPE=V(NONPRIME);
   END
```

This case statement sets TYPE to one of three status values, depending on the value of the integer item NUM. If NUM is outside of the range from 1 to 20, the status value is "V(OUT'OF'RANGE)". If NUM is in the range and is prime, the status value is "V(PRIME)". If NUM is in the range but not prime, the status value is "V(NONPRIME)". Each time the statement is executed, the value of NUM is compared to the list of values in parentheses. If it matches one of them, then the statement on that line is executed. The notation "8:10" means "8,9,10".

The case-selector (NUM in the example just given) can be an integer, bit, character, or status formula. It is not unusual for a routine to be dominated by a single case-statement, and case-statements are often nested within larger case-statements.

Loop-statements are used to repeat a sequence of statements. Here is an example of a loop-statement:

```
FOR I:0 BY 1 WHILE I<1000;
   BEGIN
   VAL = INPUT;
   IF VAL < 0;
      EXIT;
   GIVEN(I) = VAL;
   END
```

This statement uses the function INPUT to get an input value and assigns that value to VAL. It assigns input values to GIVEN(1), GIVEN(2), GIVEN(3), and so on until either GIVEN(999) has been assigned or a negative input is encountered. The examples uses an EXIT statement, which causes immediate exit from the enclosing loop.

JOVIAL also has a form of loop that has just the WHILE clause; it can be used when the loop does not require an index. Many calculations can be written as a while loop (which keeps going until some end condition is met) that encloses a case-statement (which selects the proper action for each time through the loop).

JOVIAL has GO TO statements and optional statement labels to go with them. Many programmers avoid using GO TO statements and labels, in accordance with current programming style; but they are there when needed.

Finally, JOVIAL has a STOP statement. Its meaning depends on the particular implementation; but its purpose is to provide a controlled return to the program environment.

1.1.7 Subroutines

A JOVIAL program is a collection of subroutines that are grouped together in a way described later in this introduction. Ideally, these subroutines are small. When a given subroutine gets too big, part of it is pulled out and made into a separate subroutine. In this way, each subroutine is small enough to be understood, improved, tested, and, later in the life of the program, modified.

A subroutine can be either a procedure, which is called in a procedure-call-statement, or a function, which returns a value and is used in a formula.

Here is an example of a procedure:

```
PROC RETRIEVE(CODE:VALUE);
   BEGIN
   ITEM CODE U;
   ITEM VALUE F;
   VALUE = -99999.;
   FOR I:Ø BY 1 WHILE I<1ØØØ;
      IF CODE = TABCODE(I);
         BEGIN
         VALUE = TABVALUE(I);
         EXIT;
         END
  END
```

The procedure RETRIEVE has one input parameter CODE and one output parameter VALUE. If the value of CODE is found in the global table to which the entry TABCODE belongs, the associated value TABLVALUE is returned. If the value of CODE is not found, the value -99999. is returned.

This procedure could be written as a function, as follows:

```
     PROC FIND(CODE) F;
        BEGIN
        ITEM CODE U;
        FIND = -99999.;
        FOR I:Ø BY 1 WHILE I<1ØØØ;
           IF CODE = TABCODE(I);
              BEGIN
              FIND = TABVALUE(I);
              EXIT;
              END
       END
```

The function FIND has an input parameter CODE and a return value, which is designated within the function by the function-name FIND.

The following assignment statement has the same result as a procedure-call-statement on RETRIEVE.

```
    VALUE = FIND(CODE);
```

The function FIND returns either the value associated with the value of CODE in the table or the value -99999. indicating that the value of CODE was not found.

In these examples, the search took place in a global table with 1000 entries. The subroutines can be written to accept a table of any length as a parameter. Here is the function FIND rewritten to search a table provided as a parameter:

```
PROC FIND(CODE,TAB);
   BEGIN
   ITEM CODE U;
   TABLE TAB(*);
      BEGIN
      ITEM TABCODE U;
      ITEM TABVALUE F;
      END
   FIND = -99999.;
   FOR I:0 BY 1 WHILE I<UBOUND(TAB,0);
      IF CODE = TABCODE(I);
         BEGIN
         FIND = TABVALUE(I);
         EXIT;
         END
  END
```

This function accepts the table to be searched as an actual parameter. The declaration of the table formal parameter uses the * character to indicate that the bounds are to be taken from the bounds of the table given as the actual parameter. The built-in function UBOUND, then, is used in the loop-statement to control the number of times the loop is executed.

Subroutines can also be recursive or reentrant. A recursive subroutine must have the attribute REC in its declaration and a reentrant subroutine must have the attribute RENT in its declaration.

1.1.8 Programs

A program is made up of modules. A module is a separately compilable portion of a program. A program must have one, and only one, main program module. It can have any number of procedure and compool modules.

The main program module contains the actions to be performed in the program. Execution of the program starts at the first statement in the main program module and continues until either a stop-statement or the last statement in the main program module is reached.

A procedure module contains procedures that are to be shared. Consider the following procedure module, which contains an external declaration for the subroutine FIND:

```
START
   !COMPOOL 'DATA';
   DEF PROC FIND(CODE,TAB);
      BEGIN
      ITEM CODE U;
      TABLE TAB(*);
         BEGIN
         ITEM TABCODE U;
         ITEM TABVALUE F;
         END
      FIND = -99999.;
      FOR I:Ø BY 1 WHILE I<UBOUND(TAB,Ø);
         IF CODE = TABCODE(I);
            BEGIN
            FIND = TABVALUE(I);
            EXIT;
            END
     END
TERM
```

The procedure module begins with the reserved word START and ends with the reserved word TERM. It contains a compool-directive that provides a link with the compool module DATA and an external subroutine definition (indicated by the reserved word DEF).

The reserved word DEF indicates that a data declaration or subroutine definition is external and can, therefore, be used in other modules. The reserved word REF indicates that a data declaration or subroutine definition is an external whose corresponding DEF specification is given in another module.

A compool module contains information that is to be shared:

```
START COMPOOL DATA;
   DEF TABLE PRIVILEGE(100);
          BEGIN
          ITEM NUMBER U;
          ITEM RATING F;
          END
   DEF TABLE ASSIGNMENT(999);
          BEGIN
          ITEM KEY U;
          ITEM COORDINATE F;
          END
   DEF ITEM LIMIT U;
   REF PROC FIND(CODE,TAB) F;
          BEGIN
          ITEM CODE U;
          TABLE TAB(*);
             BEGIN
             ITEM TABCODE U;
             ITEM TABVALUE F;
             END
          END
TERM
```

The compool DATA contains three external data declarations (DEF specifications) and an external subroutine reference (REF specification).

An example of a main program module using these procedure and compool modules is:

```
START !COMPOOL ('DATA');
   PROGRAM MAIN;
     BEGIN
     FOR I:0 BY 1 WHILE I < UBOUND(PRIVILEGE,0);
       IF FIND(I,PRIVILEGE) = FIND(I**2,ASSIGNMENT);
          STOP 21;
     STOP 22;
     END
TERM
```

This main program module uses the tables declared in the compool module and the function FIND defined in the procedure module and referenced in the compool module. The program consists of the main program module, the compool module DATA and the procedure module.

1.1.9 Compiler Directives

Compiler directives give information to the compiler about how to interpret and process the program. The previous section introduced the compool-directive, which provides for sharing data between modules. Other directives supply information to the compiler about optimization, register control, listing format, conditional compilation, tracing, and the like.

- For Module Linkage:

```
!COMPOOL 'C1' (AA,BB);
!LINKAGE FORTRAN;
```

- For Optimization:

```
!LEFTRIGHT;
!REARRANGE;
!ORDER;
!INTERFERENCE XX:YY;
!REDUCIBLE;
```

- For Register Control:

```
!BASE X'ITEM 2;
!ISBASE X'ITEM 2;
!DROP 2;
```

- For Listing Options:

```
!LIST;
!NOLIST;
!EJECT;
```

- For Conditional Compilation:

```
!BEGIN A;
!END;
!SKIP A;
```

- Miscellaneous:

```
!COPY 'INSERT';
!TRACE XX;
!INITIALIZE;
```

1.1.10 Compiler Macros

The define capability of JOVIAL (J73) allows the definition and use of macros. Here is an example of a simple macro:

```
DEFINE REDALERT "CONDITION=V(RED) AND STATIONS=V(CALLED)";
```

The define-name REDALERT is associated with the define-string shown above in double quotes. When a define-name is given in the text of a program, the compiler substitutes the associated define-string. For example, consider the following statement:

```
IF REDALERT;
   BATTLEPLAN(1);
```

The compiler substitutes the define-string for the define-name REDALERT to get the following statement:

```
IF CONDITION=V(RED) AND STATIONS=V(CALLED);
   BATTLEPLAN(1);
```

Macros are convenient because they permit a succint representations that can be easily modified. They are powerful because they can be used in a structured way to develop a specialized language.

The define capability of JOVIAL (J73) also permits the use of parameters in macros. In addition, list controls can be specified in the define-declaration that determine whether the macro is to be shown in its macro form, its expanded form, or both.

1.1.11 Advanced Features

The advanced features of JOVIAL (J73) allow the programmer to exercise control over the way in which data is represented and allocated. If the programmer does not specify positioning and allocation, the compiler performs these tasks. In some cases, however, the positioning must be nonstandard to allow for communication with a device that requires a particular format.

Data positioning is accomplished by specified tables and allocation by the overlay-declaration. A specified table is a table in which the programmer supplies the starting bit and starting word of each item. The overlay-declaration lets the programmer specify the allocation order of data, the machine address at which to allocate the data, or a physical overlay of data.

1.2 IMPLEMENTATION DEPENDENT CHARACTERISTICS

Each implementation of JOVIAL(J73) has special characteristics. The implementation parameters of JOVIAL help the programmer to write programs that can be machine independent. For example, the implementation parameter BITSINWORD gives the number of bits in a word for a given implementation. The information that pertains to a particular implementation of JOVIAL (J73), such as the values of the implementation parameters and the character set, is given in the user's guide for that implementation.

1.3 OUTLINE OF THIS MANUAL

The first four chapters of this manual provide a general view of the structure of a program. These chapters are:

1. Introduction
2. Program Elements
3. Program Structure
4. Declarations and Scopes

The chapter on "Program Elements" describes the characters and symbols from which a JOVIAL (J73) program is constructed; thus it is concerned with the smallest units of structure. In contrast, the chapter on "Program Structure" describes the largest units of structure, the program itself and the modules that make up the program. Finally, the chapter on "Declarations and Scopes" describes a different kind of structure, namely the assignment of meanings to names through declarations.

The next five chapters of the manual are concerned with the declaration of data. These chapters are:

5. Data Declarations
6. Item Declarations
7. Table Declarations
8. Block Declarations
9. Type Declarations

The chapter on "Data Declarations" discusses the data objects of JOVIAL (73) in a general way. The next three chapters describe specific kinds of data. The chapter on "Type Declarations" describes a way to give a name to a data type description and then use that name in the declaration of data. Type declarations support the use of pointers.

The next three chapters describe the calculation of values. These chapters are:

10. Data References
11. Formulas
12. Built-In Functions
13. Conversion

The chapter on "Formulas" describes formulas in general, dealing with operands, operators, and operator precedence. Then it describes the formulas for integer, float, fixed, bit, character, status, pointer, and table values. Finally, it describes formulas that can be calculated at compile time. The chapter on "Built-In Functions" gives, for each built-in function, the form of the function call and examples of its use. The chapter on "Conversion" describes the conversion operators and the contexts in which conversion can occur. The next chapter describes all the executable statements of JOVIAL (J73). It is:

14. Statements

This chapter begins with the assignment statement and continues with control statements. The latter include statements for conditional branching, two forms of iteration, unconditional transfer, procedure invocation, and various forms of exit.

The next chapter describes the definition and call of procedures and functions. It is:

15. Subroutines

This chapter also describes the inline-declaration, which directs the compiler to replace a subroutine call by the body of the subroutine itself rather than by a jump to the subroutine.

The next two chapters describe the way modules are put together to make a program. These chapters are:

16. Modules and Externals
17. Directives

The chapter on "Modules and Externals" describes the three different kinds of modules and the use of external names for communication between modules. The chapter on "Directives" describes a facility for including instructions to the JOVIAL (J73) compiler within a program.

The next two chapters describe special features of JOVIAL (J73). These chapters are:

18. Define Capability
19. Advanced Topics

The chapter on the "Define Capability" describes the macro facility of JOVIAL (J73). The chapter on "Advanced Topics" describes the layout of tables in storage, the overlay declaration, specified status lists, and DEF-block instantiations.

The appendixes to this manual provide reference information. They are:

A. Language Summary
B. Implementation Parameters

The "Language Summary" contains a complete syntax of JOVIAL (73). The appendix on "Implementation Parameters" describes the parameters which specialize a program for a particular computer, and which can be changed when the program is moved.

1.4 SUGGESTIONS TO THE READER

Probably you have read most of the introduction. From that, you should have an idea of the scope and power of JOVIAL. If you have worked with other high order languages, you know which features of JOVIAL are familiar to you and which are not.

Now you probably should read through the remaining chapters of the manual, not stopping to study, but just getting an idea of how the information is organized. There is more than one way to describe any language, and you need to know how this manual is put together.

If you have not worked with some form of syntactic notation before, you may find the syntax of Appendix A obscure. In that case, let it go for a while. The complete syntax given in Appendix A becomes more useful when you have learned some of JOVIAL and done some programming. Then you will have specific, detailed questions about JOVIAL, and you should find the Appendix useful.

Chapter 2

PROGRAM ELEMENTS

At the simplest level of structure, a JOVIAL (J73) module is just a sequence of characters. These characters are the letters, digits, and punctuation marks that are normally used for computer input/output.

Consider the following example, which is a fragment of a JOVIAL (J73) program:

```
SPEED3=2Ø;
```

This example is a sequence of ten characters. It begins with the five letters "S", "P", "E", "E", and "D". The letters are followed by the digit "3". Next comes the mark "=". After that is a sequence of two digits, "2" and "Ø". The sequence concludes with the mark ";".

At the next level of structure, beyond characters, a program is a sequence of symbols. Each symbol is a sequence of one or more characters that is interpreted as a single construct.

As an example, consider again the program fragment that was used to illustrate characters:

```
SPEED3=2Ø;
```

The ten characters of this example form four symbols. The characters "SPEED3" form a symbol that is the name of a variable. The single character "=" is a symbol that indicates assignment of a value to a variable. The digits "2Ø" are the symbol for the number twenty. And, finally, the character ";" is the symbol that marks the end of this construct (which is an assignment statement).

The first two sections of this chapter define characters and symbols, respectively. The third section describes the use of blanks and new lines to make a program module readable.

This chapter lays the foundation for the following chapters. It describes the symbols from which the larger constructs of JOVIAL (J73), such as formulas, statements, and entire modules, are built.

2.1 CHARACTERS

A JOVIAL (J73) character is a letter, a digit, a mark, or a special character. These characters are described in the following paragraphs.

2.1.1 Letters

JOVIAL (J73) programs can be written entirely in upper case letters. If lower case letters are available in a given implementation, they can be used. However, a lower case character is considered to be identical to the corresponding upper case character unless it appears in a character literal (defined later in this chapter).

For example, consider the following three names:

    ABC Abc abc

These names are equivalent in JOVIAL (J73). In contrast, consider the following character literals:

    'ABC' 'Abc' 'abc'

These literals are not identical in JOVIAL (J73) because the distinction between upper and lower case is retained.

2.1.2 Digits

JOVIAL (J73) uses the ten digit characters, namely:

    Ø 1 2 3 4 5 6 7 8 9

2.1.3 Marks

In describing JOVIAL (J73), the word "mark" is used to describe a character that is used as an operator, delimiter, or separator. The blank character is a mark. In addition, the following characters are marks:

+ - * / < > = @ . : , ; ( ) ' " % ! $

In some environments, certain marks are not available. In each such case, a standard alternative character is defined. A complete list of the alternative characters is given in Appendix A.

2.1.4 Special Characters

The set of special characters varies from one implementation of JOVIAL (J73) to another. These characters can be used in character literals. They have no other role in the language, but they may have a special purpose in a particular implementation. Part of the documentation of a particular implementation of JOVIAL (J73) is a list of its special characters.

2.2 SYMBOLS

The JOVIAL (J73) characters are combined to form JOVIAL (J73) symbols. The different kinds of symbols are:

Kind of Symbol	Examples
Name	VERSION AZIMUTH
Reserved Word	CASE IF GOTO
Operator	+ - **
Literal	2 3.14159 'GREY WIRE'
Status Constant	V(RED) V(CASE)
Comment	% Input Preparation Routine %
Define-String	"!A+!B"
Define-Call	TALLY(COUNT)
Index-Letter	I J
Separator	, ;

2.2.1 Names

A name is a sequence of letters, digits, dollar signs, and primes. It must begin with a letter or a dollar sign, and it must be at least two characters long. A symbol composed of a single character is not a name; instead, it is an "index letter" and is used in conjunction with loop-statements.

The following are all valid JOVIAL (J73) names:

```
ALPHA     AA      $STATUS    PART'NUMBER

Q$$$      $Ø165   P'''''     $'

POINT'OF'DEPARTURE'FOR'INCOMING'MESSAGES
```

A JOVIAL (J73) name can be any length, but the compiler only looks at at the first 31 characters. Thus the first 31 characters of a name must distinguish the name from all other names in the same scope. For example, the name POINT'OF'DEPARTURE'FOR'INCOMING'TRAINS is considered by JOVIAL to be the same name as POINT'OF'DEPARTURE'FOR'INCOMING'MESSAGES because the first 31 characters are the same.

In some implementations, the compiler may look at fewer than the first 31 characters of an <u>external</u> name. (An external name is one that is used for communication between modules or with the environment. These names are described in Chapter 16 on "Modules and Externals" .) The exact rule for recognizing external names is documented in the user's guide for the implementation.

A dollar sign in a name is translated to an implementation-dependent representation. For example, suppose a given JOVIAL (J73) system requires that each external name be prefixed by a period. The use of the period in a JOVIAL (J73) name is not allowed, but the dollar sign can be used for this purpose. The given system can then translate dollar sign to period to obtain a valid external name.

A prime (') can be used where a blank character (which is not allowed in a name) would be used, as in the following names:

```
INITIAL'TIME  FINAL'TIME  RATE'OF'DESCENT
```

2.2.2 Reserved Words

A reserved word is a symbol that has special meaning in the JOVIAL (J73) language. Reserved words are used as keywords in statements and as built-in function names. They cannot be used as names.

For example, the following are reserved words:

IF   CASE   ABS   BIT   ITEM

A complete list of the reserved words and their meanings is given in Appendix A.

2.2.3 Operators

Operators are used in JOVIAL (J73) formulas. The operators are:

Classification	Operators
Arithmetic	+ - * / ** MOD
Bit	NOT AND OR XOR EQV
Relational	= <> < <= > >=
Dereference	@
Assignment	=

The arithmetic, bit, and relational operators have their usual meanings. They are described in Chapter 11 on "Formulas".

The assignment operator is used in the assignment statement, as described in Chapter 14 on "Statements".

The dereference operator is used to obtain the object referred to by a pointer, as described in Chapter 10 on "Data References".

2.2.4 Separators

A separator is used between list elements or logical parts of a statement. It is also used to terminate statements, to delimit the beginning and the end of a construct, and to mark special constructs.

For example, the following characters are separators:

```
,    ;    :    (    )    (*    *)    !
```

Consider the following procedure-call:

```
COMBINATIONS(THINGS,OCCURENCES)
```

The comma separator "," separates the arguments in the parameter list. The parentheses delimit the parameter list.

A complete list of the JOVIAL (J73) separators and their purpose in the language is given in Appendix A.

2.2.5 Literals

A literal is a data object whose value and type are inherent in the form of its representation. JOVIAL (J73) has the following kinds of literal:

- Integer
- Real
- Bit
- Boolean
- Character
- Pointer

The different kinds of literals are described in the following paragraphs.

2.2.5.1 Integer Literals

An integer literal is a sequence of one or more digits. It is interpreted as a decimal representation of an integer value. For example, the following are all integer literals:

```
25      39876     77
```

The type of an integer literal is a signed integer type with size equal to the multiple of BITSINWORD -1 used to represent the minimum number of bits necessary to represent the value of the literal. BITSINWORD is the implementation parameter that gives the number of bits in a word for a given implementation.

For example, the minimum number of bits necessary to represent the value 25 is 5. The size, n, of the integer literal 25, thus is 15 if BITSINWORD is 16.

Note that an integer (or real) literal can be preceded by a sign, but the sign is an operator (see Chapter 11 on "Formulas") and not part of the literal.

2.2.5.2 Real Literals

A real literal is one of the following:

decimal number
decimal number followed by exponent
integer number followed by exponent

A decimal number is a sequence of digits that contains a decimal point somewhere. An integer number is a sequence of digits that does not contain a decimal point. An exponent is the letter "E" followed by an optionally signed integer number. No blank character is permitted within a real literal.

Examples of real literals are:

67.2 7853.21E-2 25E5 1E0 .003E003

A real literal can be interpreted as a floating or fixed type. The way in which it is interpreted depends on its context. The rules for its interpretation are given in Chapter 13 on "Conversion".

2.2.5.3 Bit Literals

A bit literal represents a bit string value. A bit literal is composed of a string of beads. The number of bits in each bead is given at the beginning of the bit literal as bead-size. The form of a bit literal is:

bead-size B ' bead ... '

This form uses some special notation. The "..." after "bead" means "a sequence of one or more beads". Blanks are not permitted anywhere in a bit literal.

The bead-size of a bit literal can be 1 through 5. Bead can be any digit or any letter from A through V. The digits Ø through 9 represent their actual values; the letters A through V represent the values 1Ø through 31, respectively.

The beads specified in the bit literal must be consistent with the specified bead-size. For example, the bead A, which requires four bits for its representation, cannot be used in a bit literal that has a bead-size of 3.

An example of a bit literal is:

    4B'1ØAC6'

Since the bead-size is 4, this bit literal is in hexadecimal notation. It is equivalent to the following bit literal:

    1B'ØØØ1ØØØØ1Ø1Ø11ØØØ11Ø'

In this representation, the bead size is 1, so the bit literal is in binary notation.

The size of a bit literal is the product of the bead-size and the number of bits in a bead. The size of both of the the bit literals given above is 4*5 = 2Ø bits.

2.2.5.4 Boolean Literals

A Boolean literal represents a truth value. A Boolean literal can be either TRUE or FALSE. TRUE is equivalent to the bit literal 1B'1' and FALSE to the bit literal 1B'Ø'.

2.2.5.5 Character Literals

A character literal is a string of characters enclosed in single-quote characters. The form is:

    ' character ... '

The sequence "..." means that one or more characters can be given.

The following are character literals:

```
'ABCDEFG'   'RED GREEN BLUE'   'Greetings'   '2+2=4'
```

Each blank within the delimiting single-quotes counts as a character of the character literal. The size of the character literal is the number of characters within the enclosing single-quotes. For example:

Character-Literal	Size
`' ABC '`	5
`' ABC'`	4
`'ABC'`	3
`' '`	1

A single-quote mark is represented in a character literal by two single-quote marks, and this pair counts as a single character. Two single-quote marks indicate to the compiler that the character literal has not yet come to an end. An example is:

```
'Say ''Hello'''
```

This character literal represents the three characters "Say", followed by a blank character, followed by just one single-quote character, followed by the five characters "Hello", followed by one single-quote character. Thus the entire literal represents a sequence of 11 characters.

2.2.5.6 Pointer Literals

JOVIAL (J73) has just one pointer literal, namely:

```
NULL
```

Any pointer item, regardless of whether it is typed or untyped, can be set to NULL. A typed pointer is one that is declared with an associated type-name, as described in the section on "Pointer Type Descriptions" in Chapter 6.

2.2.6 Comments

A comment is a sequence of characters enclosed in a pair of double-quotes or a pair of percent signs. Thus the forms are:

```
" character ... "

% character ... %
```

A double-quote cannot be used within a comment that is enclosed in double-quotes but a percent character can appear. For example,

```
"Applies in only 10% of the cases"
```

Similarly, a percent cannot be used within a comment that is enclosed in percents, but a double-quote can appear. For example:

```
%For details, see standard publication "Formatting"%
```

2.2.7 Other Symbols

The following symbols are described later in this manual:

Symbol	Reference
Define-String	Chapter 18
Define-Call	Chapter 18
Index-Letter	Chapter 14

The define-string and define-call symbols are used to implement macros. The index-letter is used as a loop-variable in a loop-statement.

2.3 PROGRAM FORMAT

Space characters and new lines can be used between symbols to determine the format of program listings. The compiler ignores this format, but programmers depend on good format to help them understand the structure of a program.

2.3.1 Space Characters

Space characters, or blanks, can be used between the symbols of a JOVIAL (J73) program. Using blanks, the same statement can be written in several different ways. For example:

```
IMPACT = 2Ø * HEIGHT ;

IMPACT = 2Ø*HEIGHT;

IMPACT=2Ø*HEIGHT;
```

These statements are all equivalent. Under varying circumstances, each of them might be selected as "more readable" than the others.

A blank can appear in a character literal, a comment, a define string, or a status-constant. A blank cannot appear in any other symbol. For example, consider the following assignment statement:

```
S2 = 'Press HALT';
```

This statement has three blanks in it. The first two, before and after '=', are between symbols and therefore only affect readability. The third one, inside the character literal, represents the sixth character in that literal, and is just as significant as the surrounding letters.

Now suppose a blank is inserted after "S" in the assignment statement just discussed. It becomes:

```
S 2 = 'Press HALT';
```

The insertion of a blank into the name "S2" breaks it into two symbols, the letter "S" and the integer literal "2". That changes the interpretation of the example and, in this case, produces an invalid statement.

2.3.2 New Lines

A <u>new line</u> can be used between symbols to improve readability of a program and, of course, to keep the lines to a manageable size. Like the blank character, a new line can also be used in a comment or a define string; but, unlike the blank character, a new line cannot be used in a character literal.

The way in which a new line is stored in a program file depends on the implementation and environment of JOVIAL (J73). In one implementation it may be a carriage-return and line-feed, in another it may be an end of record.

2.3.3 Formatting Conventions

The use of blanks and new lines together allows statements to be formatted. For example, you can write an if-statement in the following way, using blanks and new lines:

```
IF COND = V(RED);
   COUNT1 = COUNT1 + 1;
ELSE
   COUNT2 = COUNT2 + 1;
```

The formatting makes the logic of the statement clear to the reader.

The examples given in this manual follow formatting conventions that have been found useful by some programmers. It is difficult or even unwise to lay down strict conventions. Such rules are difficult to express and sometimes interfere with legitimate differences of style between programmers.

Some general suggestions are:

1. If a construct occupies more than one line, indent the middle lines relative to the first and last lines. (Some programmers indent the last line, too, leaving only the first line unindented.)

2. Use blanks between the main constructs of a line, and omit them (where possible) from high priority operators. Thus, for example, the assignment:

   ```
   ALPHA = 2*B + 1;
   ```

3. Similarly, use blank lines between the main constructs of the program modules.

4. Use comments, but place them so that they do not obscure the indentation structure of the program.

Chapter 3

PROGRAM STRUCTURE

A program has structure. Not only can it have several modules, but each module can be divided into parts, and those parts can, in turn, be divided into smaller parts. For example, a module can contain a subroutine definition, which can contain a statement, which can contain a formula. A formula can contain yet another, smaller formula. Ultimately, each formula is made up of symbols, such as: names, numbers and operators.

This chapter describes the largest parts of a program. The first section describes the construction of the program itself from modules. The second section describes modules in general and the "main program module" in particular.

Subsequent chapters describe the smaller components of a program -- the statements, declarations, and so on -- that are used to build modules.

3.1 THE PROGRAM

A program is a collection of one or more modules. Each module is created and maintained as a separate text file. The modules are compiled separately and then linked together for execution as a unit. The details of compilation, linking, and execution are different for each implementation of JOVIAL (J73), and are given in the user's guide for each implementation.

3.2 MODULES

A module is a sequence of symbols separated, where necessary, by blank characters and new lines.

Special constructs, the directives, can be inserted at various places in a module. Directives are an advanced feature of JOVIAL (J73) that provide instructions for the compiler. They are described in Chapter 17 on "Directives".

JOVIAL (J73) has three kinds of module, as follows:

Main Program Module -- A program must have exactly one main program module. Execution of the program begins with this module.

Procedure Module -- A program can have any number of procedure modules, or none at all. A procedure module contains data and subroutines that could be in the main program module, but that are placed in a separate module to improve organization of the program.

Compool Module -- A program can have any number of compool-modules, or none at all. A compool module contains declarations that are shared among other modules.

Procedure modules and compool modules help in the development of large programs in several ways.

1. When one module is changed and the others are not, only the changed module and the modules it affects need be recompiled.

2. If the size of the main program module exceeds the capacity of the compiler, a portion of it can be removed and embodied in a new procedure module. After that, each of the resulting modules is smaller and more likely to fit the compiler.

3. When a large project is organized, each program module can be assigned to a specific programmer. Thus program organization can parallel staff organization.

4. Certain modules can be shared among projects. Thus general libraries can be developed for a JOVIAL (J73) installation.

The description of procedure modules and compool modules is easier to understand after the other features of JOVIAL (J73) have been described. Therefore these modules are described much later in this manual, in Chapter 16 on "Modules and Externals".

Any JOVIAL (J73) program of modest size can be written as just a main program module, without any other modules. The description of the main program module follows.

3.2.1 The Main Program Module

The main program module combines declarations of data, executable statements, and subroutine definitions in a single file that can be compiled, linked to other modules (if necessary), and executed.

The form of the main program module is:

```
START PROGRAM name
   BEGIN

   [ declaration  ... ]

   statement ...

   [ subroutine-definition  ... ]

   END

   [ subroutine-definition  ... ]

TERM
```

In this representation of a main program module, the "..." under "declaration" means "a sequence of declarations", and the notation has a analagous meaning with "statement" and "subroutine-declaration". The square brackets around the declarations and subroutine-definitions indicate that these constructs can be omitted.

The symbols given in upper case in the form are JOVIAL (J73) reserved words. The words in lower case are defined as follows:

name -- This name (just after PROGRAM) is the name of the program. It is used by the JOVIAL (J73) environment in referring to the program; specific details are implementation dependent.

declaration -- The declarations after BEGIN are optional. If the body of the module does not require declarations, none need appear here. On the other hand, each name used in the module and not otherwise declared must be declared here. Declarations are described in the next chapter.

statement -- At least one statement is required. Otherwise, the main program module, and the program as a whole, would do nothing. In most cases, these statements exercise overall control of the program; that is, they invoke subroutines that do most of the work. The statements are described in Chapter 14 on "Statements".

subroutine-definition -- Subroutine-definitions can appear in two places, before and after the END. They are optional in both places; if subroutines are needed, then they must be included. The subroutine-definitions before the END are called "nested", and those after END are called "non-nested". Only non-nested subroutines can be designated as external by the use of the reserved word DEF. External subroutines are described in Chapter 16 on "Externals and Modules". Subroutine-definitions are described in Chapter 15 on "Subroutines".

Execution of the entire program begins with execution of the first statement of the main program block. Execution proceeds from one statement to the next, except where redirected by a subroutine-call, an if-statement, or some other control-statement. Execution of the program is complete when the last statement of the main program block has been executed. (There are other ways to exit a program, but that is the only way to complete it.)

Chapter 4

DECLARATIONS AND SCOPES

The main program module, described in the previous chapter, contains declarations. The other kinds of modules, the procedure module and the compool module, also contain declarations. In fact, declarations are an important part of a JOVIAL (J73) program.

A declaration is a "non-executable" construct. That is, it does not represent an action taken when the program is executed. Instead of causing action, each declaration provides information about a name that is used in the program. That information is used by the compiler each time it encounters a use of the declared name.

A declaration does not, in most cases, extend over the entire program. Instead, it applies to a particular part of the program, called the "scope" of the declaration. In fact, the same name can be declared more than once in a program, and each declaration will apply only to its scope. Thus you do not need to worry about conflicts of names in unrelated parts of a program.

The first section of this chapter describes features that all declarations have in common and then lists the different kinds of declarations. The second section describes the scopes to which declarations apply.

4.1 DECLARATIONS

A declaration always begins with a reserved word that specifies the purpose of the name being declared. For example, a declaration that begins with the reserved word ITEM specifies that the name being declared designates storage for a scalar data value (a JOVIAL item).

Once the purpose of the name has been established, the declaration provides further specialized information. As an example, consider the following declaration:

```
ITEM VELOCITY S 15;
```

This declaration declares the name VELOCITY. The reserved word ITEM means that VELOCITY is the name of storage for a scalar data value; or, to use the technical language of JOVIAL (J73), VELOCITY "designates a data object". This declaration also gives some specialized information about VELOCITY. The "S" means that it is a signed integer, and the "15" means that it occupies fifteen bits in addition to the sign.

As a program is compiled, the compiler refers back to the information obtained from the declaration of the name each time it encounters a use of that name. For example, consider the following assignment statement:

```
VELOCITY = 3;
```

In order to process this statement, the compiler must know the type of VELOCITY; that is, its type-class and how many bits are allocated for its absolute value. That information must come from a declaration of VELOCITY.

4.1.1 The Classification of Declarations

A declaration is one of the following:

Data-Declaration -- This construct declares a variable or constant name; that is, a name that designates a data object. Data-declarations are described in the next chapter.

Type-Declaration -- This construct declares a name that can be used in a data-declaration or conversion operator as an abbreviation for a data description. Type-declarations are described in Chapter 9.

Subroutine-Declaration -- This construct declares the name of a subroutine. It describes the parameters of the subroutine and (if the subroutine is a function) the result. It may also give certain special attributes of the subroutine itself. Subroutine-declarations are described in Chapter 15.

Statement-Name-Declaration -- This construct declares the name of a statement; that is, a label. Labels are usually defined by the label field in a statement; this declaration is only required for certain labels. Statement-name-declarations and the circumstances under which they are required are described in Chapter 15 on "Subroutines".

Define-Declaration -- This construct declares a name that can be used as an abbreviation for a string of JOVIAL (J73) text. Thus it provides a limited macro facility for use within a program. Define-Declarations are described in Chapter 18.

External-Declaration -- This construct declares a name that can be used in more than one module. By this means, both subroutines and data can be shared among modules. External declarations are described in Chapter 16 on "Modules and Externals".

Overlay-Declaration -- This construct establishes a relationship between previously declared data object names. It can specify names that designate the same data object or it can give the absolute address of a data object. Overlay-declarations are described in Chapter 19 on "Advanced Topics".

Inline-declaration -- This construct directs the compiler to replace a subroutine call on a given subroutine by an inline compilation of the subroutine body instead of by a transfer to the subroutine. Inline-declarations are described in Chapter 15 on "Subroutines".

Readonly-declaration -- This construct informs the compiler that the data within a subroutine is readonly and any attempt to change the values of the data is an error. Readonly-declarations are described in Chapter 15 on "Subroutines".

Null and Compound Declarations -- These declarations are special constructs that make adjustments in the syntax of declarations. They are described later in this chapter.

4.1.2 The Null-Declaration

The null declaration has the form:

```
;
```

That is, it is just a semicolon. You need this declaration when the syntax calls for a sequence of one or more declarations, but you have no names to declare. This case arises in the declaration of a subroutine that does not have parameters.

4.1.3 The Compound-Declaration

The compound-declaration has the form:

```
BEGIN
declaration
...
END
```

The sequence "..." indicates that one or more declarations can be given within a BEGIN-END pair.

The sequence of declarations can be empty, so that a special form of the compound declaration is:

```
BEGIN
END
```

Compound-declarations enable a group of declarations to be treated syntactically as a single declaration.

4.2 SCOPE

Each declaration in a program supplies information about a particular name. However, a given declaration of a given name does not necessarily apply to all occurrences of that name. The occurrences of a name to which a declaration does apply is the scope of that declaration.

Scopes are established during the compilation of a module.

A system-scope and a compool-scope enclose the module being compiled. These scopes can be diagrammed as follows:

```
+system-scope-----------------------------------------+
|                                                     |
|    +compool-scope-------------------------------+   |
|    |                                            |   |
|    |    +module-scope----------------------+    |   |
|    |    |                                  |    |   |
|    |    |                                  |    |   |
|    |    |                                  |    |   |
|    |    |                                  |    |   |
|    |    |                                  |    |   |
|    |    +----------------------------------+    |   |
|    |                                            |   |
|    +--------------------------------------------+   |
|                                                     |
+-----------------------------------------------------+
```

The compool scope and the system scope are not actually part of the source file for the module being compiled, but they can be thought of that way.

All names made available from referenced compool modules, as well as the names of the compools themselves, belong to the compool scope. In addition, the name of the module being compiled is itself considered to belong to this outer compool scope. External names and compools are described in Chapter 16 "Modules and Externals". More examples of scope are also given in that chapter.

System-defined names, such as implementation-parameters and machine-specific subroutines belong to the system scope. Such names can be redefined in the enclosed scopes.

The module being compiled is a scope and has smaller scopes within it. The module scope contains the names of any non-nested subrouines. Within the module-scope, the module-body establishes a scope. It contains the names declared within the module-body. Within the module-body, subroutine-bodies establish scopes and within subroutine-bodies, other subroutine-bodies establish scope, and so on. Ultimately, there are scopes that do not themselves contain further scopes.

The scope of a module thus can be diagrammed as follows:

```
+system-scope-------------------------------------------------+
|                                                             |
|     +compool-scope---------------------------------+       |
|     |                                              |       |
|     |    +module-scope-------------------------+  |       |
|     |    |                                     |  |       |
|     |    |   +module-body-scope----------+    |  |       |
|     |    |   |                            |    |  |       |
|     |    |   |   +subr-scope-----------+ |    |  |       |
|     |    |   |   |                     | |    |  |       |
|     |    |   |   |  +subr-scope------+ | |    |  |       |
|     |    |   |   |  |                | | |    |  |       |
|     |    |   |   |  |  +subr-scope--+ | | |    |  |       |
|     |    |   |   |  |  |  ...        | | | |    |  |       |
|     |    |   |   |  |  +------------+ | | |    |  |       |
|     |    |   |   |  +----------------+ | |    |  |       |
|     |    |   |   +---------------------+ |    |  |       |
|     |    |   |                            |    |  |       |
|     |    |   +----------------------------+    |  |       |
|     |    |                                     |  |       |
|     |    |   +subr-body------------------+    |  |       |
|     |    |   |                            |    |  |       |
|     |    |   |   +subr-scope-----------+ |    |  |       |
|     |    |   |   |                     | |    |  |       |
|     |    |   |   |  +subr-scope------+ | |    |  |       |
|     |    |   |   |  |                | | |    |  |       |
|     |    |   |   |  |  +subr-scope--+ | | |    |  |       |
|     |    |   |   |  |  |  ...        | | | |    |  |       |
|     |    |   |   |  |  +------------+ | | |    |  |       |
|     |    |   |   |  +----------------+ | |    |  |       |
|     |    |   |   +---------------------+ |    |  |       |
|     |    |   +----------------------------+    |  |       |
|     |    |     ...                             |  |       |
|     |    |                                     |  |       |
|     |    +-------------------------------------+  |       |
|     |                                              |       |
|     +----------------------------------------------+       |
|                                                             |
+-------------------------------------------------------------+
```

For example, consider the following main-program-modul^:

```
START PROGRAM TEST;
 +module-scope-------------------------------------------------+
 |    +module-body-scope--------------------------------------+ |
 |    | BEGIN                                                  | |
 |    |    ITEM LENGTH U;                                      | |
 |    |     ...                                                | |
 |    |                                                        | |
 |    |    PROC CALCULATE (OP1,OP2:RESULT);                    | |
 |    |    +subr-scope-------------------------------+         | |
 |    |    |    BEGIN                                |         | |
 |    |    |    ITEM OP1 F;                          |         | |
 |    |    |    ITEM OP2 F;                          |         | |
 |    |    |    ITEM RESULT F;                       |         | |
 |    |    |    ITEM SIZE U;                         |         | |
 |    |    |    ...                                  |         | |
 |    |    |                                         |         | |
 |    |    |    END                                  |         | |
 |    |    +-----------------------------------------+         | |
 |    |    PROC COMPUTE (OP1,OP2:RESULT);                      | |
 |    |    +subr-scope-------------------------------+         | |
 |    |    |    BEGIN                                |         | |
 |    |    |    ITEM OP1 F;                          |         | |
 |    |    |    ITEM OP2 F;                          |         | |
 |    |    |    ITEM RESULT F;                       |         | |
 |    |    |    ITEM SIZE U;                         |         | |
 |    |    |    ...                                  |         | |
 |    |    |                                         |         | |
 |    |    |    PROC SUBTOTAL(TOTAL:RESULT);         |         | |
 |    |    |    +subr-scope-------------------+      |         | |
 |    |    |    |  BEGIN                      |      |         | |
 |    |    |    |  ITEM TOTAL U;              |      |         | |
 |    |    |    |  ITEM RESULT U;             |      |         | |
 |    |    |    |  ...                        |      |         | |
 |    |    |    |                             |      |         | |
 |    |    |    |                             |      |         | |
 |    |    |    |  END                        |      |         | |
 |    |    |    +-----------------------------+      |         | |
 |    |    |    END                                  |         | |
 |    |    +-----------------------------------------+         | |
 |    |   END                                                  | |
 |    +--------------------------------------------------------+ |
 |    DEF PROC REPORT(IN,OUT);                                   |
 |        ...                                                    |
 +-------------------------------------------------------------+
TERM
```

In addition to the system scope and the compool scope, five additional scopes are defined. The lines in the above program indicate the scopes. The scope of the module TEST encloses the scope of the program-body, which encloses the scopes of the procedures CALCULATE and COMPUTE. The scope of the procedure COMPUTE encloses the scope of the procedure SUBTOTAL.

The scopes of the module TEST can be diagrammed as follows:

```
+system-scope-----------------------------------------+
|                                                     |
|    +compool-scope-------------------------------+   |
|    |       TEST                                 |   |
|    |   +module-scope-----------------------+    |   |
|    |   |                                   |    |   |
|    |   |  +module-body-scope----------+    |    |   |
|    |   |  | LENGTH                     |    |    |   |
|    |   |  | CALCULATE                  |    |    |   |
|    |   |  |  +subr-scope-y--------+    |    |    |   |
|    |   |  |  | OP1,OP2,RESULT,SIZE|    |    |    |   |
|    |   |  |  |                    |    |    |    |   |
|    |   |  |  +--------------------+    |    |    |   |
|    |   |  |  COMPUTE                   |    |    |   |
|    |   |  |  +subr-scope----------+    |    |    |   |
|    |   |  |  |OP1,OP2,RESULT,SIZE |    |    |    |   |
|    |   |  |  |SUBTOTAL            |    |    |    |   |
|    |   |  |  | +program-body----+ |    |    |    |   |
|    |   |  |  | |TOTAL, RESULT   | |    |    |    |   |
|    |   |  |  | |                | |    |    |    |   |
|    |   |  |  | +----------------+ |    |    |    |   |
|    |   |  |  +--------------------+    |    |    |   |
|    |   |  |                            |    |    |   |
|    |   |  |                            |    |    |   |
|    |   |  |                            |    |    |   |
|    |   |  +----------------------------+    |    |   |
|    |   |  REPORT                           |    |   |
|    |   |  +subr-scope-----------+          |    |   |
|    |   |  | IN, OUT             |          |    |   |
|    |   |  +---------------------+          |    |   |
|    |   +-----------------------------------+    |   |
|    |                                            |   |
|    +--------------------------------------------+   |
|                                                     |
+-----------------------------------------------------+
```

The item SIZE and the procedure names CALCULATE and COMPUTE are in the scope of the procedure module. The names OP1 and OP2 are in the scope of both CALCULATE and COMPUTE. The name RESULT is in the scope of CALCULATE, COMPUTE, and SUBTOTAL. A reference to RESULT within SUBTOTAL refers to an output parameter of SUBTOTAL that is an unsigned integer. A reference to RESULT within COMPUTE refers to an output parameter of COMPUTE that is a floating object.

4.2.1 The Scope of a Declaration

The scope of a declaration is the smallest scope that contains the declaration.

Each use of a name must have a declaration. That declaration is determined as follows:

1. If the reference to the given name does not lie in the scope of any declaration of that name, then the program is invalid.

2. If the reference to the given name lies in the scope of exactly one declaration of the given name, then that declaration applies to the given use of the name.

3. If the reference to the given name lies in the scope of several declarations of that name, then the declaration with smallest scope applies to the given use of the name.

With these definitions in mind, consider another version of the example given earlier in this chapter. This example includes references to names.

```
START PROGRAM TEST;
 +module-scope-----------------------------------------------+
 |   +module-body-scope-------------------------------------+ |
 |   | BEGIN                                                | |
 |   |     ITEM LENGTH U;                                   | |
 |   |      ...                                             | |
 |   |                                                      | |
 |   |     PROC CALCULATE (OP1,OP2:RESULT);                 | |
 |   |     +subr-scope----------------------------+         | |
 |   |     |   BEGIN                              |         | |
 |   |     |   ITEM OP1 F;                        |         | |
 |   |     |   ITEM OP2 F;                        |         | |
 |   |     |   ITEM RESULT F;                     |         | |
 |   |     |   ITEM SIZE U;                       |         | |
 |   |     |   ...                                |         | |
 |   |     |   LENGTH = 21;                       |         | |
 |   |     |   ...                                |         | |
 |   |     |   END                                |         | |
 |   |     +--------------------------------------+         | |
 |   |     PROC COMPUTE (OP1,OP2:RESULT);                   | |
 |   |     +subr-scope----------------------------+         | |
 |   |     |   BEGIN                              |         | |
 |   |     |   ITEM OP1 F;                        |         | |
 |   |     |   ITEM OP2 F;                        |         | |
 |   |     |   ITEM RESULT F;                     |         | |
 |   |     |   ITEM SIZE U;                       |         | |
 |   |     |   ...                                |         | |
 |   |     |                                      |         | |
 |   |     |   PROC SUBTOTAL(TOTAL:RESULT);       |         | |
 |   |     |   +subr-scope---------------------+  |         | |
 |   |     |   |  BEGIN                        |  |         | |
 |   |     |   |  ITEM TOTAL U;                |  |         | |
 |   |     |   |  ITEM RESULT U;               |  |         | |
 |   |     |   |  ...                          |  |         | |
 |   |     |   |  RESULT = TOTAL**2;           |  |         | |
 |   |     |   |  ...                          |  |         | |
 |   |     |   |                               |  |         | |
 |   |     |   |  END                          |  |         | |
 |   |     |   +-------------------------------+  |         | |
 |   |     |   END                                |         | |
 |   |     +--------------------------------------+         | |
 |   |    END                                               | |
 |   +------------------------------------------------------+ |
 |   DEF PROC REPORT(IN,OUT);                                 |
 |       ...                                                  |
 +------------------------------------------------------------+
TERM
```

The reference to LENGTH lies within the scope of exactly one delcaration of that name. The reference to RESULT lies within the scope of two declarations of that name. In this case, the declaration given in the procedure SUBTOTAL, which is the smaller scope, applies.

4.2.2 Restrictions on Declarations

The following restrictions apply to the declaration of names:

1. Two declarations of the same name must not have the same scope.

2. A reserved word must not be used as a name and cannot, therefore, be declared. The reserved words are listed in Appendix A.

3. An external name must be declared by exactly one DEF declaration in an entire program. External declarations are described in Chapter 16.

Chapter 5

DATA DECLARATIONS

A data-declaration declares a variable-name or a constant-name. A variable-name designates storage for a value that can be changed during program execution. A constant-name can be thought of as designating storage for data that is set before program execution and then does not change; in many cases, however, actual storage is not required for the value of a constant-name.

JOVIAL (J73) provides abstract storage. Storage is ultimately implemented as a hardware memory composed of words, bytes, and bits that have numeric addresses. However, JOVIAL (J73) can screen out the irrelevant hardware details and present you with a more convenient and logical storage structure.

Although you can, when necessary, specify an absolute storage address, the association of storage addresses is normally handled for you by the compiler. Although you can, when necessary, request that a variable be a specific word of hardware memory, you normally describe the kind of values you want to store and let the compiler allocate the correct amount of storage at an appropriate location. JOVIAL (J73) permits you to ignore hardware details when they are not important but lets you specify them in considerable detail when considerations of efficiency and interfacing require.

In order to emphasize this treatment of storage, this manual uses the term <u>data object</u> to refer to the storage for a value or a collection of values. You can talk of "fetching the value of a data object" or "assigning a value to a data object" without any knowledge of the implementation of the data object.

The first section of this chapter introduces the three kinds of data declarations. The second and third sections make distinctions that apply to all data objects: the difference between variable and constant values and between automatic and static allocation.

5.1 THE CLASSIFICATION OF DATA DECLARATIONS

A data-declaration is one of the following:

Item-Declaration -- This construct declares the name of a scalar data object; that is, storage for a single value. Item-declarations are described in Chapter 6.

Table-Declaration -- This construct declares the name of a table data object; that is, a collection of items. Table-Declarations are described in Chapter 7.

Block-Declaration -- This construct declares the name of a block data object; that is, a collection of items and tables. Block declarations are described in Chapter 8.

5.2 VARIABLES AND CONSTANTS

A data object can be variable or constant. In an item-declaration or table-declaration, the reserved word CONSTANT means that the declared name designates a constant data object. This reserved word may be placed at the beginning of any item-declaration or table-declaration, as described in the next two chapters. The absence of the reserved word CONSTANT means the declared name designates a data object that is variable.

5.2.1 Variable Data Objects

A variable data object has a relatively complicated life cycle. First, it is allocated; that is, a portion of storage is set aside to hold the value of the data object. As the following section shows, allocation can occur either during or before program execution.

After the variable data object has been allocated, it can be initialized. Initialization occurs if the variable name is declared with a preset, as described in the next three chapters on item-declarations, table-declarations, and block-declarations. Initialization also can occur through an initialize-directive, as described in Chapter 16 on "Directives". If neither a preset nor a directive applies, the data object is not initialized.

Next, the variable data object is used; that is, values are assigned to it and fetched from it. If the data object was not initialized, its first use must be as a target of an assignment or as an output parameter of a subroutine. The value of a variable data object that does not have an initial value and has not yet been assigned a value is undefined.

When the execution of the program of procedure that declares the variable is complete, the variable data object is deallocated; that is, the storage associated with the data object is taken away.

5.2.2 Constant Data Objects

A constant data object has a simple life cycle. The data object has the same value throughout program execution. That value is supplied by the same item-preset or table-preset mechanism that is used for initializing variable data objects. In some cases, the value of a constant data object may require storage in data memory, but in many cases the value is embedded into the code of the compiled program. A program that attempts to assign a value to a constant data object is invalid.

5.3 STORAGE ALLOCATION

The allocation of a variable data object can be automatic or static. A data object has automatic allocation only if it is declared in a subroutine body and its declaration does not have the STATIC reserved word. A data object has static allocation if it is not in a subroutine body or it has the STATIC reserved word. The STATIC reserved word is placed immediately after the variable name that is being declared.

5.3.1 Automatic Allocation

For automatic allocation, the storage for a data object is allocated and deallocated when the subroutine in which the data object is declared is entered and exited. Automatic allocation saves storage by holding it only during execution of a subroutine. However, the value of such a data object is lost upon exit from a subroutine, and is therefore undefined upon each entry to the subroutine.

5.3.2 Static Allocation

For static allocation, the storage for a data object is allocated before program execution and is deallocated, at earliest, when program execution is complete. Even when a static data object is declared within a subroutine, its value is retained from one execution of the subroutine to the next.

Chapter 6

ITEM DECLARATIONS

An item is a scalar variable or constant. This chapter describes the general form of the item declaration, first for a variable and then for a constant. It then considers the different data types of JOVIAL J73.

6.1 ITEM DECLARATIONS

An item-declaration specifies that an item-name designates a variable or constant with a given type class and attributes. The simplest form of an item-declaration is:

```
ITEM  item-name  type-description ;
```

This form declares an item-name that designates a scalar variable of the type given by type-description.

For example, consider the following item-declaration:

```
ITEM COUNT U 5;
```

Item-name in this declaration is COUNT and type-description is U 5. This declaration specifies that COUNT designates a scalar variable with the type U 5. The U indicates that the item is an unsigned (that is, non-negative) integer. The 5 specifies that the integer occupies 5 bits. Data types are described in detail later in this chapter.

An item-declaration can also give information about the allocation permanence of the variable and its initial value, as follows:

```
ITEM  item-name [ STATIC ] type-description [ item-preset ] ;
```

The square brackets indicate that STATIC and item-preset are both optional.

Consider the following item-declaration:

```
ITEM COUNT U 5 = Ø;
```

This declaration specifies that COUNT designates an unsigned integer variable with the initial value Ø. Item-preset, in this case, consists of an equals sign and the initial value Ø. Item-presets are discussed in detail later in this chapter.

The STATIC attribute is provided so that items within subroutines can have static allocation. The default allocation permanence of data within subroutines is automatic.

Consider the following item-declaration:

```
ITEM COUNT STATIC U 5;
```

This declaration specifies that COUNT designates an unsigned integer variable with STATIC allocation permanence.

6.2 CONSTANT ITEM DECLARATIONS

A constant item-declaration is a special form of an item-declaration. It begins with the reserved word CONSTANT and concludes with an item-preset. A constant item receives its value before the execution of the program. The value of a constant item cannot change during the program execution.

The form of a constant item-declaration is:

```
CONSTANT ITEM item-name type-description item-preset ;
```

The allocation permanence of all constants, even those within procedures, is STATIC. Physical storage is allocated for all constant declarations given in a block. For constants not declared in a block, physical storage may not be allocated if another technique for representing the constant can be used in the code generated by the compiler.

As an example of a constant item-declaration, consider the following:

```
CONSTANT ITEM VERSION U = 22;
```

This declaration specifies that VERSION designates an unsigned integer constant whose value is (always) 22. Throughout the scope of this declaration, VERSION can be used anywhere the integer literal 22 could be used.

6.3 DATA TYPES

A data type consists of a type class and a set of attributes. The scalar type classes are:

Unsigned Integer
Signed Integer
Floating
Fixed
Bit
Character
Status
Pointer

In an integer, floating, or fixed type-description, the number of bits occupied is given, either explicitly or by default. The number of bits is interpreted by the compiler as the minimum storage requirement. If it is advantageous for the compiler to use more bits, it may do so.

The largest (in magnitude) value that such an item can have, however, is determined by the number of bits given or assumed in its declaration, not the number of bits actually used by the compiler. If a value that cannot be accomodated in the number of bits given or assumed in its declaration is assigned to an item, then the program is invalid.

Type-description in an item-declaration describes the type of the item. The following sections describe and illustrate each type-description.

6.3.1 Integer Type-Descriptions

An integer is a signed or unsigned value that occupies a specified number of bits. Type-description for an unsigned integer has the form:

```
U [ integer-size ]
```

Type-description for a signed integer has the form:

```
S [ integer-size ]
```

The square brackets indicate that integer-size is optional.

Integer-size is an integer compile-time-formula. A compile-time-formula is a formula whose value can be determined at compile time. Compile-time-formulas are discussed in Chapter 11 on "Formulas". Integer-size determines the minimum number of bits allocated by the compiler for the integer; it determines the maximum value that can be accommodated by the item. The compiler allocates at least integer-size bits for an unsigned integer and at least (integer-size + 1) bits for a signed integer.

Integer-size must lie in the range:

Ø < integer-size ≤ MAXINTSIZE

MAXINTSIZE is the implementation parameter that defines the maximum size of an integer.

The range of values that an integer can assume is machine dependent. A signed integer can take on values in the range:

MININT (integer-size) ≤ value ≤ MAXINT(integer-size)

An unsigned integer variable can take on values in the range:

Ø ≤ value ≤ MAXINT(integer-size)

MAXINT and MININT are the implementation parameters that define the maximum and minimum values of an integer.

Some examples of integer item-declarations are:

Declaration	Meaning
ITEM TIME U 5;	TIME designates an unsigned 5-bit integer variable. It occupies a minimum of 5 bits. If it is declared in a subroutine, it has automatic allocation; otherwise, it has static allocation. Its initial value is unspecified. It can assume the values Ø through MAXINT(5).
ITEM RANGE S 1Ø;	RANGE designates a signed 1Ø bit integer variable. It occupies a minimum of 11 bits. It can assume values in the range MININT(1Ø) through MAXINT(1Ø).
ITEM POSITION U;	POSITION designates an unsigned integer variable. If BITSINWORD is 16, it occupies a minimum of 15 bits. It can assume the values Ø through MAXINT(15).
ITEM COUNT STATIC U 1Ø;	COUNT designates an unsigned 1Ø-bit integer variable with STATIC allocation permanence.
ITEM TIME U 5 = 2Ø;	TIME designates an unsigned 5-bit integer variable with an initial value of 2Ø.
CONSTANT ITEM LIMIT U = 1Ø;	LIMIT designates an unsigned integer constant with the value 1Ø.

6.3.2 Floating Type-Descriptions

A floating value is expressed as a mantissa and an exponent. The form of a floating type-description is:

```
F [ precision ]
```

The square brackets indicate that precision is optional.

Precision is an integer compile-time-formula that determines the minimum number of bits allocated for the mantissa of the floating-point value. The total storage required for the item is always more than the precision, because the representation must include storage for the exponent and sign. Furthermore, the compiler may allocate more bits than required.

Precision must lie in the range:

0 < precision ≤ MAXFLOATPRECISION

MAXFLOATPRECISION is the implementation parameter that determines the maximum precision of a floating type. If no precision is given, the compiler uses the implementation parameter FLOATPRECISION as the precision.

A variable of floating type can assume the following values:

MINFLOAT(precision) ≤ value ≤ MAXFLOAT(precision)

MINFLOAT and MAXFLOAT are implementation parameters.

Some examples of floating item-declarations are:

Declaration	Meaning
ITEM AZIMUTH F 30;	AZIMUTH designates a floating variable with precision 30. Its mantissa occupies a minimum of 30 bits. If it is declared within a procedure, it has automatic allocation; otherwise, it has static allocation. It is not initialized.
ITEM VELOCITY F;	VELOCITY designates a floating variable. If FLOATPRECISION is 12, its mantissa occupies 12 bits.
ITEM DISTANCE STATIC F 24 = .001;	DISTANCE designates a floating variable with precision 24, static allocation, and initial value .001.
CONSTANT ITEM COEFFICIENT F = 21.36;	COEFFICIENT designates a floating constant with the value 21.36.

6.3.3 Fixed Type-Descriptions

A fixed number is a real number with a fixed decimal point. Fixed point representation is used for numbers whose value range is known to lie within a given, usually small, range. Fixed point representation can be used for numbers that are either very large or very small and for which only a certain number of significant digits are required.

A fixed value has a fixed scale factor. Its interpretation is described by two specifiers, scale and fraction. These specifiers determine the position of the point and the number of digits, as described in the next paragraph.

A fixed type-description has the following form:

```
A  scale [ , fraction ]
```

The square brackets indicate that the fraction is optional.

When scale and fraction are both positive, scale gives the number of bits to the left of the binary point (excluding the sign bit) and fraction gives the number of bits to the right of the binary point.

For example, suppose you give the following declaration:

```
ITEM AMOUNT A 11,3;
```

AMOUNT designates a fixed point variable with eleven bits to the left of the binary point, 3 bits to the right of the binary point, and 1 bit for the sign. That is, it is laid out as follows:

```
S XXXXXXXXXXX.XXX     where X indicates a bit of storage
                      and   S indicates the sign
```

The minimum number of bits allocated for AMOUNT is 15.

If scale is negative, the binary point is assumed to be the specified number of bits to the left of the first (non-sign) bit of the representation. For example:

```
ITEM COORD A -3,9;
```

COORD designates a variable that requires at least 7 bits (-3+9+1) of storage. The binary point is assumed to be three bits to the left of first bit of the stored value. That is:

```
S .ØØØXXXXXX     where X indicates a bit of storage
```

If fraction is negative, the binary point is assumed to be the specified number of bits to the right of the least significant bit of representation. For example:

```
ITEM LIMIT A 15,-5;
```

The variable LIMIT requires at least 11 bits (15-5+1) of storage. The binary point is assumed to be five bits to the right of the last bit of the representation. That is:

```
S XXXXXXXXXXØØØØØ.   where X indicates a bit of storage
```

If fraction is not given, then the compiler assumes that the precision of the item is the implementation parameter FIXEDPRECISION and the fraction is FIXEDPRECISION minus the scale.

For example, suppose you write:

```
ITEM FACTOR A 12;
```

If FIXEDPRECISION is 15, then the precision of FACTOR is 15 and the default fraction is 3. That is:

```
S  XXXXXXXXXXXX.XXX
```

Scale and fraction are integer compile-time-formulas. The value of the scale must lie in the following range:

$-127 \le \text{scale} \le 127.$

The precision of a fixed point number is the sum of scale and fraction. The implemented precision may be greater than the declared precision. However, as mentioned earlier, the values set for a fixed point item is determined by the declared precision.

The precision must lie in the range:

Ø < scale+fraction ≤ MAXFIXEDPRECISION

MAXFIXEDPRECISION is the implementation parameter that determines the maximum precision of a fixed type.

A variable of fixed type can assume values in the range:

MINFIXED(scale,fraction) ≤ value ≤ MAXFIXED(scale,fraction)

Some additional examples of fixed item-declarations are:

Declaration	Meaning
ITEM SUBTOTAL A 6,2;	SUBTOTAL designates a fixed variable with scale 6 and fraction 2.
ITEM TICKS STATIC A 7,4 = 2.5;	TICKS is a fixed item with scale 7 and fractional part 4. It has static allocation and the initial value 2.5.
CONSTANT ITEM THRESHOLD A 1Ø,1 = 1Ø16.5;	THRESHOLD is a fixed constant with the value 1Ø16.5.

6.3.4 Bit Type-Descriptions

A bit item is a fixed length string of bits. The form of a bit type-description is:

B [bit-size]

The square brackets indicate that bit-size is optional.

Bit-size is an integer compile-time-formula that indicates how many bits are in the bit string. It must lie in the range:

1 ≤ bit-size ≤ MAXBITS

MAXBITS is the implementation parameter that defines the maximum number of bits a bit string can occupy.

If bit-size is not given, the compiler assumes the number of bits in the string to be 1.

Some examples of the declaration of bit items are:

Declaration	Meaning
ITEM MASK B 10;	MASK designates a bit variable 10 bits long.
ITEM FLAG B;	FLAG designates a bit variable 1 bit long.
ITEM READY STATIC B 3 = 1B'000';	READY designates a bit variable 3 bits long with static allocation and an initial value of all zero bits.
CONSTANT ITEM SWITCH B = TRUE;	SWITCH designates a bit constant that has the value TRUE (1B'1').

6.3.5 Character Type-Descriptions

A character item is a fixed length string of characters. The form of a character type-description is:

    C [char-size]

The square brackets indicate that char-size is optional.

Char-size is an integer compile-time-formula. Char-size must lie in the following range:

$1 \le$ char-size $\le$ MAXBYTES

MAXBYTES is the implementation parameter that defines the maximum number of characters a character string can occupy.

If char-size is not given, the compiler assumes that the number of characters in the string is 1.

Some examples of the declaration of character items are:

Declaration	Meaning
ITEM ADDRESS C 26;	ADDRESS designates a character variable 26 characters long. If it is declared within a procedure, it has automatic allocation; otherwise, it has static. Its initial value is unspecified.
ITEM CODE C;	CODE designates a character variable 1 character long.
ITEM RESPONSE STATIC C 9 = 'NOT READY';	RESPONSE designates a character variable 9 characters long with static allocation and an initial value of 'NOT READY'.
CONSTANT ITEM TITLE C 6 = 'JOVIAL';	TITLE designates a character constant with the value 'JOVIAL'.

6.3.6 Status Type-Descriptions

A <u>status item</u> is an item whose value range is a specified list of symbolic names, called <u>status-constants</u>. A status-constant is a symbolic constant that has an ordering relation with the other status constants in the list. A status constant provides an efficient way to express values symbolically.

The simplest form of a status type-description is the reserved word STATUS followed by a parenthesized list of status constants, as follows:

```
STATUS ( status-constant ,... )
```

The sequence ",..." indicates that one or more status-constants separated by commas can be given within the parentheses.

The form of a status-constant is the letter "V" followed by a parenthesized status-name, as follows:

```
V ( status-name )
```

A status-name can be a name, a letter, or a JOVIAL (J73) reserved word.

The use of a name in a status-constant does not constitute a declaration of the name or a reference to a declared name with the same spelling. For example, the status-constant V(MONDAY) declares the name V(MONDAY), not MONDAY. A status-name and a declared name with the same spelling can exist in the same scope without any conflict.

The status-constants are represented as the values Ø through N-1, where N is the number of status-constants in the list. The values Ø through N-1 are the default representations of the status-constants; that is, the compiler uses these values if a specific representation is not given in the declaration. This form of STATUS type-description is described in Chapter 19 on "Advanced Topics".

Suppose you write:

```
STATUS  (V(RED),V(GREEN),V(BLUE),V(YELLOW));
```

The status list contains four status constants. The first constant V(RED) is represented as the value Ø, the second V(GREEN) is represented as 1, and so on.

Even though the representation of a status-constant is an integer, an integer value cannot be assigned to a status item unless it is first explicitly converted to a status type.

The size of a status item is the minimum number of bits necessary to hold the representation of the status-constant with the largest reprentation.

Another form of the status type-description allows both the representation of status-constants and the specification of the status size to be given. This form is described in Chapter 19 on "Advanced Topics".

The representation of a status-constant determines its order for relational operators. That is, for the declaration given above, the status-constants have the following relationship:

```
V(RED) < V(GREEN) < V(BLUE) < V(YELLOW)
```

The names in any given status-list must be unique. However, the same name can be given in more than one list. In most cases, any ambiguity is resolved by the context in which the status-constant is used. Sometimes, however, an explicit conversion operator must be used to make the status-constant unambiguous. Chapter 13 on "Conversion" discusses both these cases.

Some examples of status items are:

Declaration	Meaning
ITEM DAY STATUS (V(SUN), V(MON), V(TUES), V(WED), V(THURS), V(FRI), V(SAT));	DAY designates a status variable that can take on the values V(SUN) through V(SAT).
ITEM IVY STATIC STATUS (V(ENGLISH),V(GRAPE), V(POISON)) = V(ENGLISH);	IVY designates a status variable that can take on 3 values. It has static allocation and is initialized to the status-constant V(ENGLISH).
CONSTANT ITEM ID STATUS (V(DESIGN),V(DEBUG),V(RUN)) = V(RUN);	ID designates a status constant with the value V(RUN).

6.3.7 Pointer Type-Descriptions

A pointer item is used to locate data. The values of a pointer item are addresses of objects.

The form of a pointer type-description is:

```
P [ type-name ]
```

The square brackets indicate that type-name is optional.

A pointer that is declared without type-name is called an untyped pointer. A pointer that is declared with type-name is called a typed pointer. Type-names are declared by a type-declaration, as described in Chapter 9 on "Type Declarations". Typed pointers can point only to objects that are declared in terms of the same type-name.

Pointers are used with items, tables, and blocks that are declared using a type-name. The pointer makes the referenes to the data objects unambiguous. Pointers, however, must be used when referencing data declared using a type, even if no ambiguity is involved.

Some examples of pointer items are:

Declaration	Meaning
ITEM PTR P;	PTR is an untyped pointer.
ITEM P1 P PARTS;	P1 is a typed pointer; it can point only to objects of type PARTS.
CONSTANT ITEM PDATA P SEQ = LOC(DATA);	PDATA designates a pointer constant with value of LOC(DATA). DATA must be of type SEQ.

6.4 ITEM-PRESETS

The item-preset provides an initial-value for an item-declaration and a permanent value for a constant item-declaration. The form of an item-preset is:

= value

Value must be a compile-time-formula for all items except pointers. A pointer can have a LOC function as an initial value. (The LOC function is a built-in function that gets the address of a data object. It is described in Chapter 12). However, if the argument of a LOC function used in a preset is a data-name, the data-name must designate an item with STATIC allocation permanence.

The value in an item-preset is assigned to the item before program execution. It must be compatible with the type of the item, as defined by the type-description. The rules for type compatibility are given in Chapter 13, "Conversion".

For example, the following item-declarations all contain valid item-presets:

```
ITEM COUNT U 10 = 0;
ITEM AZIMUTH F = .01;
ITEM BALANCE A 12,2 = 15.25;
ITEM MASK B 5 = 1B'00100';
ITEM ID C 5 = 'AWC  ';
ITEM CODE STATUS (V(HAWK), V(WOLF), V(TIGER), V(SNOOPY))
                 = V(WOLF);
ITEM PTR P = NULL;
```

6.4.1 The Round-or-Truncate Attribute

The type-description of an integer, floating, or fixed type can contain a round-or-truncate attribute. The round-or-truncate attribute is given following the single letter that identifies the type class. It is separated from that letter by a comma. The forms are:

```
U [ , round-or-truncate ]

S [ , round-or-truncate ]

F [ , round-or-truncate ]

A [ , round-or-truncate ]
```

The square brackets indicate that the round-or-truncate attribute is optional.

The round-or-truncate attribute is either an R or a T. The attribute R indicates that rounding occurs when the value in the preset is assigned to the item. The attribute T indicates that truncation toward minus infinity occurs.

If a round-or-truncate attribute is not given, truncation in a machine-dependent manner occurs. Truncation may be either towards zero or towards minus infinity depending on the implementation.

A round-or-truncate attribute in an integer type-description has no purpose in an item-declaration. However, consider the following floating item-declaration:

```
ITEM VELOCITY F 10 = 1200.3;
```

In some implementations, the value of 1200.3 may not be exactly representable as a floating point value (e.g. in a binary number system) and thus it must be rounded or truncated. This declaration does not include a round-or-truncate attribute, so the value is truncated in a machine dependent manner. However, if you wish the value to be rounded, you can add a round attribute as follows:

```
ITEM VELOCITY F,R 10 = 1200.3;
```

The preset value is then rounded before assignment to VELOCITY.

The round-or-truncate attribute is most useful in a type-description used as a conversion operator, as will be seen in Chapter 13 on "Conversion".

Chapter 7

TABLE DECLARATIONS

A JOVIAL (J73) table is a collection of data objects. A table can be dimensioned or undimensioned. An undimensioned table has only one entry. A dimensioned table is made up of one or more entries.

The form of a table-declaration is:

```
TABLE  table-name  [ table-attributes  ] ;

       entry-description
```

The square brackets indicate that table-attributes can be omitted.

The following is an example of a table-declaration:

```
TABLE MATRIX(1:20);
   BEGIN
   ITEM XCOORD U;
   ITEM YCOORD U;
   END
```

This declaration declares a table named MATRIX. Table-attributes in this declaration is (1:20), indicating that MATRIX has 20 entries. The first entry is referenced as MATRIX(1) and the last entry as MATRIX(20).

Entry-description indicates that each entry in the table contains two items, XCOORD and YCOORD. The instance of the item XCOORD in the first entry is referenced as XCOORD(1).

7.1 TABLE-ATTRIBUTES

Table-attributes gives the attributes of the table. It can specify the allocation permanence of the table, indicate whether or not the table is dimensioned, and provide any initial values for the components of the table. Table-attributes has the following form:

```
[ STATIC ] [ ( dimension-list ) ] [ table-preset ]
```

The square brackets indicate that any of the parts of table-attributes can be omitted.

Table-attributes can also contain information about the way in which the table is structured and packed. A description of table structure and packing is given in Chapter 19 on "Advanced Topics".

7.1.1 Allocation Permanence

The allocation attribute STATIC can be given in a table-attributes..

For example, suppose you declare the table STOCKS within a procedure and you want it to have STATIC allocation. You can write:

```
TABLE STOCKS STATIC (1:10);
    BEGIN
    ITEM NAME C 6;
    ITEM QUOTE C 3;
    END
```

7.1.2 Table Dimensions

The table dimensions are given in dimension-list. The dimensions of a table specify the number of entries in the table and the number of subscripts required in a reference to an item in the table.

Dimension-list is a sequence of one or more dimensions, as follows:

```
dimension ,...
```

A table can have as many as seven dimensions. Entries are arranged in a table so that the rightmost subscripts vary fastest, from lower-bound to upper-bound.

7.1.2.1 Bounds

For each dimension of a table a lower-bound and upper-bound can be given. The form of a dimension is:

```
[ lower-bound : ] upper-bound
```

The square brackets indicate that lower-bound is optional.

Each bound must be a compile-time-formula of either status or integer type. Only status types with default representations can be used as bounds. Lower-bound must be less than or equal to upper-bound.

A one-dimensional table is a table for which only a single dimension is specified. For example, to declare a one-dimensional table with lower-bound 1 and upper-bound 5, you can write the following declaration:

```
TABLE TEST (1:5);
   ITEM SUCCESS U 5;
```

The table TEST contains the following five integers:

```
SUCCESS(1)
SUCCESS(2)
SUCCESS(3)
SUCCESS(4)
SUCCESS(5)
```

If only upper-bound is given, then the compiler assumes a lower-bound based on the type of upper-bound. There are two cases, one for type integer and one for type status.

If upper-bound is an integer, it must be a positive integer. In this case, the compiler assumes a lower-bound of Ø. For example, to declare a one-dimensional table with lower-bound Ø and upper-bound 5, you can write the following declaration:

```
TABLE TEST(5);
   ITEM SUCCESS U 5;
```

This table contains six entries. The first entry is the integer SUCCESS(Ø) and the sixth entry is the integer SUCCESS(5).

If upper-bound is a status value, the status value must be associated with only one status type. A status-value that is associated with more than one status type is ambiguous in this context and must be disambiguated by a conversion operator, as discussed in Chapter 13 on "Conversion".

For an unambiguous status type, the compiler assumes that lower-bound is the first status-constant in the status type of the upper-bound. For example, suppose you have the following declarations:

```
ITEM INDEX STATUS (V(IRELAND), V(ENGLAND), V(FRANCE));

TABLE VOYAGE4 (V(FRANCE));
   ITEM TIME U;
```

V(FRANCE) is a member of the status list associated with the item INDEX. The compiler assumes that lower-bound is V(IRELAND). The table VOYAGE4, therefore, has three entries. The first item is referred to as TIME(V(IRELAND)), the second as TIME(V(ENGLAND)), and the third as TIME(V(FRANCE)).

7.1.2.2 Table Size

The total number of entries in a table is calculated by multiplying the number of entries in each dimension. If the bounds of a dimension are integers, the number of entries in that dimension is found by subtracting lower-bound from upper-bound and adding 1. Suppose you have the following table declaration:

```
TABLE INSTALLATIONS (5,2:6,10:20);
   ITEM ID U;
```

The first dimension of the table INSTALLATIONS has lower-bound 0 and upper-bound 5; the second dimension has lower-bound 2 and upper-bound 6; the third dimension has lower-bound 10 and upper-bound 20. The number of entries, therefore, is:

```
(5-0+1)*(6-2+1)*(20-10+1) = 6*5*11 = 330
```

That is, the table INSTALLATIONS contains 330 entries.

If the bounds are status values, the number of entries is found by subtracting the position of lower-bound in the list of status constants from the position of upper-bound in that list and adding 1.

Suppose you have the declarations:

```
ITEM SEASON STATUS
    (V(SPRING),V(SUMMER),V(FALL),V(WINTER));

TABLE WEATHER(88,V(FALL));
    ITEM RAINFALL U;
```

The first dimension has lower-bound 0 and upper-bound 88 and thus contains 89 entries.

The second dimension has lower-bound V(SPRING) and upper-bound V(FALL). The status constant V(SPRING) is the first constant in the status list given in the declaration of SEASON and the constant V(FALL) is the third constant on that list. The second dimension, therefore, contains 3 entries.

The total number of entries in the WEATHER table, therefore, is:

```
(88-0+1)*(3-1+1) = 89*3 = 267
```

7.1.2.3 Maximum Table Size

The number of words occupied by a table must not exceed the following quotient:

MAXBITS/BITSINWORD

MAXBITS is the implementation parameter that gives the maximum value for a bit string and BITSINWORD is the implementation parameter that gives the number of bits in a word.

7.1.3 Table-Preset

The initial values that a table is automatically assigned on allocation are given by table-preset. Initializing a table is described later in this chapter after the discussion of the entry description.

7.2 ENTRY-DESCRIPTION

Entry-description describes the components that make up an entry. An entry-description can be either simple or compound.

A table with a simple entry-description does not need the BEGIN END brackets. It has only one item per entry. A table-declaration with a simple entry-description has the following form:

```
TABLE  table-name  [ table-attributes  ] ;

   table-option
```

The square brackets indicate that table-attributes is optional.

A table-option is either a table item-declaration or a null-declaration. A table item-declaration is the same as an item-declaration, except that it can have a table-preset, which sets one or more instances of the item, instead of an item-preset, which sets only one instance. Table-presets are described later in this chapter.

Consider the following example of a table with a simple entry-description:

```
TABLE TRIAL (5);
   ITEM TIME U 10;
```

This declaration declares a table TRIAL. Table-attributes in this declaration indicates that the table is dimensioned and contains 6 entries, indexed from 0 through 5. Entry-description indicates that each entry contains an unsigned ten-bit integer. The integer in the first entry is referred to as TIME(0), the integer in the second as TIME(1), and so on.

A compound entry-description encloses one or more table-options between a BEGIN END pair. The form of a table-declaration with a compound entry-description is:

```
TABLE  table-name  [ table-attributes ] ;

   BEGIN

   table-option ...

   END
```

The square brackets indicate that table-attributes is optional. The notation "..." indicates that any number of table-options can be given within the BEGIN END pair.

Table-option is either an item-declaration or a null-declaration.

Consider the following example of a table declaration with a compound entry-description:

```
TABLE SPECIFICATIONS (50);
   BEGIN
   ITEM LENGTH U 5;
   ITEM WIDTH U 9;
   ITEM HEIGHT U 5;
   END
```

This table declaration declares the table SPECIFICATIONS, which has one dimension. The entry-description in this table indicates that each entry contains three items.

This table can be diagrammed as follows:

```
                 SPECIFICATIONS(50)----------+
                 |  ...                      |
                 |                           |
          SPECIFICATIONS(1)---------+        |
       SPECIFICATIONS(0)----------+ |        |
       |                          | |        |
       |  +LENGTH-------------+   | |        |
       |  |                   |   | |        |
       |  +-------------------+   | |        |
       |                          | |        |
       |  +HEIGHT-------------+   | |        |
       |  |                   |   | |        |
       |  +-------------------+   | |        |
       |                          | |        |
       |  +WIDTH--------------+   | |--------+
       |  |                   |   | |
       |  +-------------------+   |-+
       +--------------------------+
```

The first item in the first entry is referred to as LENGTH(0), the second item in the first entry as WIDTH(0), and so on.

7.2.1 Unnamed Entry-Descriptions

One additional form of the table-declaration is allowed, namely: one with an unnamed entry-description. This form is:

```
TABLE  table-name  [ table-attributes ]

   type-description ;
```

As an example of this form, consider the following table-declaration:

```
TABLE SCORE(1000) U 5;
```

The table SCORE contains 1001 unnamed entries. These entries can be referenced as SCORE(0), SCORE(1), and so on. The type of these references, however, is table and so their use is limited.

7.3 CONSTANT TABLE DECLARATIONS

A constant table-declaration is a table-declaration preceded by the word CONSTANT, as follows:

```
     CONSTANT TABLE table-name  table-attributes ;

                    entry-description
```

Constant tables are always allocated in static storage, so table-attributes in a constant declaration does not have an allocation attribute. Any of the other table attributes, however, can be declared. A constant table can be dimensioned or undimensioned. A constant table must have some initial values, but not all the components need be initialized. A partially-initialized constant table can be used, for example, to set and reset the constant part of a variable table.

The values in a constant table cannot change during program execution. That is, a component of a constant table cannot be used in a context in which its value can be changed.

An example of a constant table is:

```
     CONSTANT TABLE THRESHOLDS(1:10);
        ITEM LEVEL U = 2,12,26,45,99,200,315,500,1000,10000;
```

The elements of a constant table cannot be changed during the course of program execution. These elements, however, cannot be used in a compile-time-formula.

7.4 TABLE INITIALIZATION

Some or all of the items in a table can be set to initial values. The set of initial values is called a table-preset. It can be given either for an item within the table or in the table-attributes.

7.4.1 Table-Presets with Item-Declarations

The table-preset for an item follows the type-description, as follows:

```
TABLE  table-name  [ table-attributes ] ;

   BEGIN

   ITEM  item-name  type-description table-preset;
   ...

   END
```

Consider the following example, which uses a table-preset in an item-declaration.

```
TABLE STOCKS(1:1Ø);
    BEGIN
    ITEM NAME C 6 = "AAA", "ACE", "ACME";
    ITEM QUOTE C 3;
    END
```

This preset initializes the first three NAME items. That is, it is equivalent to the following:

```
NAME(1)="AAA"
NAME(2)="ACE"
NAME(3)="ACME"
```

7.4.2 Table-Presets in the Table-Attributes

A table-preset for a table is given as the last part of the table-attributes, as described earlier in this chapter.

If a table-declaration contains a table-preset in its table-attributes, then no table-presets can be given for the item-declarations within the table.

Consider the following table-declaration, which includes a table-preset in the table-attributes:

```
TABLE TØ(1:3) = 1,7,2,4,3,8;
   BEGIN
   ITEM SPEED U;
   ITEM DISTANCE U;
   END
```

The above declaration is equivalent to the following declaration, which contains table-presets with the items of the table:

```
TABLE TØ(1:3);
   BEGIN
   ITEM SPEED U = 1,2,3;
   ITEM DISTANCE U = 7,4,8;
   END
```

Both versions set the items as follows:

```
SPEED(1) = 1
DISTANCE(1) = 7
SPEED(2) = 2
DISTANCE(2) = 4
SPEED(3) = 3
DISTANCE(3) = 8
```

7.4.3 Values

A table-preset consists of a list of values, as follows:

```
value ,...
```

The notation ",..." indicates that one or more values, separated by commas, can be given.

Entries within a dimensioned table are initialized in order. The first entry to be initialized is the one with the lowest value of each dimension index. The next entry is found by incrementing the rightmost index. This process continues until the rightmost index has taken on all the values in its range, then the index to the left of the rightmost is incremented and so on.

For example, suppose you have the following table:

```
TABLE GRID(1,2) = 1,2,3,4,5,6;
    ITEM HITS U;
```

The items are initialized as follows:

```
HITS(Ø,Ø) = 1
HITS(Ø,1) = 2
HITS(Ø,2) = 3
HITS(1,Ø) = 4
HITS(1,1) = 5
HITS(1,2) = 6
```

7.4.4 Omitted Values

If values are omitted in the preset, then the corresponding items are not set. An omitted value is indicated by a comma. Suppose you want to omit setting some values in the GRID table. You can write:

```
TABLE GRID(1,2) = ,2,3,,5,6;
    ITEM HITS U;
```

The items are initialized as follows:

```
HITS(Ø,Ø)        (not initialized)
HITS(Ø,1) = 2
HITS(Ø,2) = 3
HITS(1,Ø)        (not initialized)
HITS(1,1) = 5
HITS(1,2) = 6
```

7.4.5 Preset Positioner

A positioner is used to indicate the starting position for a set of one or more values. The form is:

```
POS ( index, ... ) : value, ...
```

The notation ",..." indicates that one or more indexes or values, separated by commas, can be given.

The number of indexes given within the parentheses must agree with the number of dimensions given for the table. The indexes are subscripts and must lie within the valid range given in the dimensions.

An index is either a compile-time-integer-formula or a compile-time-status-formula, depending on whether the dimensions are integer or status types.

For example, suppose you want to initialize items 1, 2, 3, 25, 26, and 3Ø. You can use the following table-preset in the table-declaration:

```
TABLE SCHOOLSYSTEM(1:1ØØ);
   ITEM CLASS'SIZE U = 16,21,24,POS(25):31,33,POS(3Ø):18;
```

The first three values are assigned to the first three items of the table (1, 2, and 3), the next two values are assigned with respect to the positioner 25 (31 and 33)and the final value with respect to the positioner 3Ø. That is, the preset sets the following items:

```
CLASS'SIZE(1)  = 16;
CLASS'SIZE(2)  = 21;
CLASS'SIZE(3)  = 24;
CLASS'SIZE(25) = 31;
CLASS'SIZE(26) = 33;
CLASS'SIZE(3Ø) = 18;
```

Suppose you have a two-dimensional table, as follows:

```
TABLE MATRIX(4,4);
   ITEM ELEMENT F;
```

You can initialize the diagonal as follows:

```
TABLE MATRIX(4,4) = POS(Ø,Ø): Ø,
                    POS(1,1): Ø,
                    POS(2,2): Ø,
                    POS(3,3): Ø,
                    POS(4,4): Ø;
   ITEM ELEMENT F;
```

7.4.6 Repetition-Counts

A repetition-count can be used in a preset to set a number of items to the same value. The form of a repetition count is:

```
repetition-count ( list-element ,... )
```

The notation ",..." indicates that one or more list-elements, separated by commas, can be given.

Repetition count is a compile-time-integer-formula that indicates how many times to repeat the list-elements within the parentheses. A list-element can be a value or a repetition-count followed by a parenthesized list, as shown above.

For example, suppose you want to set all the items of a table to zero. You can use a repetition-count in the table-preset, as follows.

```
TABLE SCHOOLSYSTEM(1:100);
   ITEM CLASS'SIZE U = 100(0);
```

You can set the first 50 items to 1 and the second 50 to 2, as follows:

```
TABLE SCHOOLSYSTEM(1:100);
   ITEM CLASS'SIZE U = 50(1),50(2);
```

You can set the odd-numbered items to 1 and the even-numbered items to 0, as follows:

```
TABLE SCHOOLSYSTEM(1:100);
   ITEM CLASS'SIZE U = 50(1,0);
```

You can set the items in SCHOOLSYSTEM in sets of 5. Suppose the first four values in each set are 26 and the fifth value is 22. You can use the following table-preset, which contains a nested repetition count.

```
TABLE SCHOOLSYSTEM(1:100);
   ITEM CLASS'SIZE U = 20(4(26),22);
```

Suppose you want to set a block of ten entries to zero, with each block beginning at various positions in the table. You can write the following declaration:

```
TABLE SCHOOLSYSTEM(1:100);
   ITEM CLASS'SIZE U = 10(0), POS(50):10(0), POS(70):10(0);
```

This declaration sets entries 1 through 10, 50 through 59, and 70 though 79 to zero.

Chapter 8

BLOCK DECLARATIONS

A block groups items, tables, and other blocks into contiguous storage. A block also gives a collection of data objects a name so that the data can be manipulated as a whole. Blocks can, for example, be passed as parameters or declared external.

8.1 BLOCK-DECLARATION

The form of a block-declaration is:

```
BLOCK  block-name ;

   block-body
```

Block-body describes the components that make up the block. Block-body can be either simple or compound. A block with a simple block-body has only one declaration:

```
BLOCK block-name ;

   declaration
```

Declaration is a data declaration or a null declaration.

A block-declaration with a compound block-body has the form:

```
BLOCK  block-name ;
   BEGIN

   block-option ...

   END
```

Block option can be a data, overlay, or null declaration. Overlay declarations are described in Chapter 19 on "Advanced Topics".

Suppose you use the three items CODE, KEY, and LIMIT in connection with the table INVENTORY and you want to ensure that the items and the table are allocated together so that your table manipulating routine can access them efficiently. Consider the following example of a block-declaration:

```
BLOCK  INVENTORY'GROUP;
   BEGIN
   ITEM CODE U;
   ITEM KEY U;
   ITEM LIMIT U;
   TABLE INVENTORY(2000);
      BEGIN
      ITEM ORDER'NUMBER U;
      ITEM VISUAL'ID C 5;
      ITEM ONHAND U;
      END
   END
```

This block can be diagrammed as follows:

```
+INVENTORY'GROUP-------------------+
|                                  |
|   +CODE------------+             |
|   |                |             |
|   +----------------+             |
|                                  |
|   +KEY-------------+             |
|   |                |             |
|   +----------------+             |
|                                  |
|   +LIMIT-----------+             |
|   |                |             |
|   +----------------+             |
|                                  |
|   +INVENTORY---------------+     |
|   |   +ORDER'NUMBER----+   |     |
|   |   |                |   |     |
|   |   +----------------+   |     |
|   |                        |     |
|   |   +VISUAL'ID-------+   |     |
|   |   |                |   |     |
|   |   +----------------+   |     |
|   |                        |     |
|   |   +ONHAND----------+   |     |
|   |   |                |   |     |
|   |   +----------------+   |     |
|   |                        |     |
|   +------------------------+     |
+----------------------------------+
```

Because the items and tables are enclosed in a block, the compiler allocates them together. However, the compiler is free to allocate the data within the block in any order. If the order of allocation within the block is important, you can preserve it by giving an order directive. The order directive is described in Chapter 16 on "Directives".

8.1.1 Nested Blocks

Since a data-declaration can be a block-declaration, block-declarations can be nested.

For example, suppose you want to specify the grouping of your data as follows:

```
BLOCK MAINGROUP;
   BEGIN
   ITEM MASTER U;
   ITEM MASTERCODE U;
   ITEM MASTERID C 5;
   TABLE MASTERTAB(10,10);
      ITEM RECORD U;
   BLOCK SUBGROUP;
      BEGIN
      ITEM MINOR U;
      ITEM MINORCODE U;
      ITEM MINORID C 5;
      TABLE MINORTAB(100,100);
         ITEM SCORE U;
      END
   END
```

The compiler allocates this data together because of the block MAINGROUP. Further, it allocates the items MINOR, MINORCODE, and MINORID with the table MINORTAB because of the block SUBGROUP.

8.1.2 Allocation Permanence

To cause a block declared within a subroutine to have static allocation, the STATIC attribute is given in the declaration of the block following block-name, as shown in the following fragment.

```
BLOCK  block-name  [ STATIC ] ;

       block-body
```

The square brackets indicate that STATIC can be omitted.

Only blocks that have static allocation, either explicitly or by default, can contain constant declarations or declarations with presets.

A data declaration within a block cannot include an allocation attribute.

8.1.3 Initial Values

Initial values can be given for the data within a block that has static allocation by giving presets with the data declarations in the block.

For example, suppose you want to give the items CODE and LIMIT in the Block INVENTORY'GROUP the initial values 0 and 180 and you want to initialize the first 180 entries of the table. You add presets to the items and table as shown:

```
BLOCK  INVENTORY'GROUP;
   BEGIN
   ITEM CODE U = 0;
   ITEM KEY U;
   ITEM LIMIT U = 180;
   TABLE INVENTORY(2000) = 180(0,"XXXXX",0);
      BEGIN
      ITEM ORDER'NUMBER U;
      ITEM VISUAL'ID C 5;
      ITEM ONHAND U;
      END
   END
```

The same rule applies to blocks within blocks. For example, suppose you want to preset the items MASTER and MINOR:

```
BLOCK MAINGROUP;
   BEGIN
   ITEM MASTER U = 22;
   ITEM MASTERCODE U;
   ITEM MASTERID C 5;
   TABLE MASTERTAB(10,10);
      ITEM RECORD U;
   BLOCK SUBGROUP;
      BEGIN
      ITEM MINOR U = 6;
      ITEM MINORCODE U;
      ITEM MINORID C 5;
      TABLE MINORTAB(100,100);
         ITEM SCORE U;
      END
   END
```

The block declaration includes presets that set MASTER to 22 and MINOR to 6.

Chapter 9

TYPE DECLARATIONS

A type-declaration declares a name for a user-defined type. The resulting name can then be used in declaring data objects in a convenient and uniform way.

A type-name is considered to be an abbreviation for its associated item-description, table-description, or block-description. It can be used in a declaration to give the type of the data name being declared or in a conversion operator to define the type to which the operand is to be converted.

9.1 TYPE-DECLARATION

A type-declaration declares a new type by associating a name with a data description. The form of type-declaration is:

```
TYPE  type-name  data-description ;
```

Data-description can be an item, table, or block description.

For example, you can declare an item type-name by giving a type-description in a type-declaration, as follows:

```
TYPE  MODIFIER  F 20;
```

You can declare a table type-name by giving a table data-description in a type-declaration, as follows:

```
TYPE  ARRAY
   TABLE (20,20);
      ITEM U POINT;
```

PRECEDING PAGE BLANK-NOT FILMED

You can declare a block type-name by giving a block data-description in a type-declaration, as follows:

```
TYPE  OUTPARS
   BLOCK
      BEGIN
      ITEM DISTANCE U;
      ITEM SPEED U;
      TABLE SIGHTINGS(1ØØ);
         BEGIN
         ITEM LONG F;
         ITEM LAT  F;
         END
      END
```

A type-name can also be declared in terms of another, previously declared, type-name. For example, an item-type-name can be declared as follows:

```
TYPE  type-name  type-name ;
```

For example, you can declare another item type-name, as follows:

```
TYPE  SECONDMOD MODIFIER;
```

Similarly, you can declare other table type-names by giving the name of a previously defined table. That is, a table type-name can be defined as follows:

```
TYPE  table-type-name  TABLE  table-type-name ;
```

For example:

```
TYPE MOREPARS TABLE OUTPARS;
```

A type-declaration does not involve the allocation of storage. It records the description of the type. When the type is used in a data-declaration, storage is allocated.

The following sections consider item, table, and block type-declarations, in detail.

9.2 ITEM TYPE-DECLARATION

An item type-declaration has the form:

```
TYPE type-name type-description ;
```

Type-descriptions were discussed in Chapter 6 on "Item Declarations".

A type-name declared in this way can be used in an item-declaration in place of type-description. This <u>alternate form</u> of item-declaration is:

```
ITEM item-name type-name ;
```

For example, suppose you frequently use a 10-bit unsigned integer for counters. You can declare the type-name COUNTER as follows:

```
TYPE COUNTER ITEM U 10;
```

Then each time you declare a counter, you can give the type-name rather than the item-description, thus:

```
ITEM CT1 COUNTER;
ITEM CT2 COUNTER;
```

CT1 and CT2 are declared to be 10-bit unsigned integers by the type-name COUNTER.

The use of a type-name in this case provides documentation about the use of the item, ensures that all counters have the same type and allows you to make a sweeping change at a later time with a minimum of effort.

9.2.1 Allocation and Initial Values

Information about allocation and initial values cannot be included in the declaration of a type-name. However, this information can be given in a declaration that uses a type-name.

For example, suppose you declare a counter CLOCKTICK within a procedure and you want it to have static allocation and an initial value of Ø. You can write:

```
ITEM CLOCKTICK STATIC COUNTER = Ø;
```

The declaration uses the type-name COUNTER, introduced in the previous section of this chapter, to declare CLOCKTICK.

9.3 TABLE TYPE DECLARATIONS

A table type-declaration has the form:

```
TYPE type-name
   TABLE [ ( dimension-list ) ] ;
      entry-description
```

The square brackets indicate that the parenthesized dimension-list is optional.

Entry-description gives the form of each entry in the table. As in a table-declaration, entries can be simple, compound, or unnamed.

A table type-name can be used in a table declaration in place of entry-description as follows:

```
TABLE table-name [ STATIC ] [ ( dimension-list ) ]
   type-name [ table-preset ] ;
```

The names supplied by a type are potentially ambiguous and must be qualified by a pointer when used.

The type-declaration capability can be used to declare a number of tables with the same structure. Suppose, for example, that you have three tables with the same structure. You can define a type-name and then declare each of the three tables in terms of that type-name. Suppose the type-declaration declares a table type-name, as follows:

```
TYPE PART
   TABLE;
      BEGIN
      ITEM PARTNUMBER U 5;
      ITEM ONHAND U 10;
      ITEM ONORDER U 10;
      END
```

You can now define tables using the type-name PART, as follows:

```
TABLE BOLTS PART;
TABLE NUTS(100) PART;
TABLE WRENCHES(10,20) PART;
```

BOLTS is a table of type PART. NUTS is a one-dimensional table, each of whose entries is of type PART. WRENCHES is a two-dimensional table, each of whose entries is of type PART.

Observe that three tables in your program now have items with the names PARTNUMBER, ONHAND, and ONORDER. To reference one of these names, you must qualify it to make it unambiguous. Qualification is achieved by the use of pointers in JOVIAL (J73).

9.3.1 Dimension and Structure

Dimensions and information about the table layout can be given in a type-declaration. A table, however, can have at most one dimension-list and one layout. Thus, if a table declared with a type-name has a dimension-list, then the type-declaration must not have a dimension-list. Conversely, if the type-declaration has a dimension-list, then the table-declaration using the type-name must not have a dimension-list.

For example, you can define a type with dimensions as follows:

```
TYPE SPECIFICATIONS
   TABLE ( 100 );
      BEGIN
      ITEM LENGTH U;
      ITEM WIDTH U;
      ITEM HEIGHT U;
      END
```

This declaration defines a type with 101 entries. Each entry contains the items LENGTH, WIDTH, and HEIGHT.

Now you can declare a table using this type-name:

```
TABLE TRUCK SPECIFICATIONS ;
```

The table TRUCK contains 101 entries as described by the type-name. You cannot, however, declare a table and include a dimension-list in the table-attributes if you use the type-name SPECIFICATIONS in the declaration.

Control over bit layouts, if necessary, can be accomplished in JOVIAL (J73). This capability is discussed in Chapter 19 on "Advanced Topics".

9.3.2 Allocation and Initial Values

As with the item type-declaration, allocation permanence and initial values cannot be given with a table type-declaration. However, this information can be given in a table declaration that uses the type-name.

For example, suppose you want to declare a ten-entry table type name with static allocation and initialized to zero. You can declare the ten-entry table as a type-name, as follows:

```
TYPE DECADE TABLE(1:10);
   ITEM EVENTS U;
```

Then you can declare the table with static allocation and initial values of zero, as follows:

```
TABLE FIRST'DECADE STATIC DECADE = 10(0);
```

9.3.3 Like-Option

A like-option permits the use a previously declared type-name in the declaration of another type-name. Like-option follows dimension-list (if present) in the type-declaration as follows:

```
    TYPE  type-name

          TABLE [ ( dimension-list ) ] [ like-option ] ;

             entry-description
```

The form of the like-option is:

```
     LIKE  table-type-name
```

Suppose you want to define two different table types. Each has a common part, namely the first five items. The additional items in each table, however, are different. You can give a type-declaration for the common part and then use the like-option in the two table type-declarations, as follows:

```
     TYPE IDENTIFICATION TABLE;
        BEGIN
        ITEM NAME C 10;
        ITEM RANK C 5;
        ITEM SERIALNUMBER C 12;
        ITEM ACTIVEFLAG B 1;
        END

     TYPE INACTIVE TABLE
        LIKE IDENTIFICATION;
        ITEM RETIREDATE C 6;

     TYPE ACTIVE TABLE
        LIKE IDENTIFICATION;
        BEGIN
        ITEM STATION C 10;
        ITEM REVIEWDATE C 6;
        END
```

The INACTIVE table type contains the items NAME, RANK, SERIALNUMBER, ACTIVEFLAG, and RETIREDATE. The ACTIVE table type contains the items NAME, RANK, SERIALNUMBER, ACTIVEFLAG, STATION, and REVIEWDATE.

9.3.3.1 Dimensions and Like-Options

If the type-name given in the like-option contains a dimension, that dimension applies to the declaration that includes the like-option.

Suppose you have the type SPECIFICATIONS with a dimension list of 1ØØ, indicating 1Ø1 entries, and another type DESCRIPTION, which references SPECIFICATIONS in a like-option, as follows:

```
TYPE SPECIFICATIONS TABLE(1ØØ);
     BEGIN
     ITEM LENGTH U 6;
     ITEM WIDTH U 6;
     ITEM HEIGHT U 6;
     END

TYPE DESCRIPTION TABLE
     LIKE SPECIFICATIONS;
     BEGIN
     ITEM WEIGHT U;
     ITEM COLOR U;
     END
```

A table declared with type DESCRIPTION contains 1Ø1 entries. Each entry contains the items LENGTH, WIDTH, HEIGHT, WEIGHT, and COLOR.

Since a table can have only one dimension list, a type-declaration with a like-option for a dimensioned type cannot contain a dimension list. Similarly, a dimensioned table cannot have a like-option for a dimensioned type.

9.4 BLOCK TYPE DECLARATIONS

A block type-declaration has the form:

```
TYPE  type-name

   BLOCK

      block-body
```

A type-name declared in this way can be used in a block-declaration in place of block-body, as follows:

```
BLOCK  block-name  [ allocation-spec ] type-name ;
```

For example, suppose you have a standard method for managing tables and associated with each table you maintain a size and flag item. You can combine this information in a block type as follows:

```
TYPE INVENTORY
   BLOCK
      BEGIN
      ITEM MAXSIZE U;
      TABLE PARTS (49);
         BEGIN
         ITEM ID C 10;
         ITEM COUNT U;
         END
      ITEM FLAG B;
      END
```

Then you can declare blocks using that type, as follows:

```
BLOCK XSTORE INVENTORY;
BLOCK YSTORE INVENTORY;
```

The blocks XSTORE and YSTORE each contain a size, flag, and table.

As with table type-declarations, the names supplied by the type are potentially ambiguous and must be qualified by a pointer when referenced.

9.4.1 Initial Values

A block-declaration that includes a type-name can include a block-preset, as follows:

```
      BLOCK  block-name  [ allocation-spec ] type-name

                         [ block-preset ] ;
```

The square brackets indicate that the allocation-spec and the block-preset are both optional.

A block-preset consists of a sequence of values. Like the table-preset, a block-preset can contain repetition counts and positioners. In addition, a block-preset can have parenthesized table-presets or block-presets. A table- or block-preset within a block is enclosed in parentheses.

Suppose you want to declare initial values for the block YSTORE. You can write:

```
     BLOCK YSTORE INVENTORY = 5Ø, (5Ø(" ",Ø)),FALSE;
```

This declaration declares a block YSTORE with an item MAXSIZE that has an initial value of 5Ø, a table whose items are set to the blank string and Ø, respectively, and a flag that is set to FALSE.

9.4.1.1 Omitted Values

An omitted value in a block-preset indicates that the corresponding item, table, or block remains uninitialized.

Suppose you want to set only the first five entries of the table in the block and you don't want to set the value of MAXSIZE or FLAG. You can write the following declaration:

```
     BLOCK XSTORE INVENTORY = ,(5(' ',Ø));
```

This declaration does not give MAXSIZE or FLAG an initial value, but it provides initial values for the items ID and COUNT for entries Ø through 4.

Chapter 1Ø

DATA REFERENCES

The way in which data is referenced depends on its declaration. Three kinds of data reference can be made, namely:

Simple
Subscripted
Pointer-qualified

The following sections discuss each kind of data reference.

1Ø.1 SIMPLE REFERENCES

A simple reference designates a data object that has only one instance. A simple reference can reference an item, a table, a block, or an item in an unsubscripted table. The form of a simple reference is the name of the declared object, as follows:

```
name
```

Consider the following declarations:

```
ITEM LENGTH U;
TABLE STATISTICS;
   BEGIN
   ITEM COUNT U;
   ITEM WEIGHT F;
   END
BLOCK PARTLIST;
   BEGIN
   ITEM DATE C 6;
   TABLE PARTS(1ØØ);
      BEGIN
      ITEM ID C 1Ø;
      ITEM INVENTORY U;
      END
   END
```

A simple reference can be made to the item LENGTH, the table STATISTICS, the items COUNT and WEIGHT, the block PARTLIST, the item DATE and the table PARTS. All these objects can be located by a such a reference.

The table PARTS, however, contains 1Ø1 entries. Each entry contains an instance of the item ID and the item INVENTORY. A reference to ID or INVENTORY, therefore, must include a subscript to indicate which instance is indicated.

1Ø.2 SUBSCRIPTED DATA REFERENCES

If a data object is declared within a dimensioned table, then there are as many instances of that object within the table as the dimensions indicate. A reference to that object must include subscripts to indicate the instance.

The form of a subscripted data reference is:

```
name ( subscript ,... )
```

The sequence ",..." indicates that one or more subscripts can be given separated by commas.

A reference to a table entry or a table item must contain the same number of subscripts as there are dimensions in the table declaration. Further, each subscript must lie within the range specified by the bounds of the dimension.

For example, consider again the table PARTS:

```
TABLE PARTS(1ØØ);
   BEGIN
   ITEM ID C 1Ø;
   ITEM INVENTORY U;
   END
```

This table has one dimension, with lower-bound Ø and upper-bound 1ØØ. A reference to ID or INVENTORY must contain a single subscript in that range. For example, a reference to the item ID in the first entry is:

```
ID(Ø)
```

As another example, consider the following declaration:

```
TABLE TRIALX(5,2:6,10:20);
   ITEM HITS U;
```

The table TRIALX has three dimensions. Thus, a reference to an item in table TRIALX must use three subscripts and the value of each subscript must lie within the range specified by the dimensions, as follows:

```
HITS(2,2,12)
```

The first subscript 2 lies within the bounds (0:5) for the first dimension. The second subscript 2 lies within the bounds (2:6) for the second dimension. The third subscript 12 lies within the bounds (10:20) for the third dimension.

10.3 QUALIFIED DATA REFERENCES

A reference can be qualified by the use of a pointer.

Qualification can always be used in referencing a name, but in some cases qualification is necessary.

If a table is declared using a type-name, the names of the components of the table are potentially ambiguous and must be qualified by a pointer when referenced.

10.3.1 Pointer-Qualified References

A pointer in JOVIAL (J73) can be used to locate a particular table and, in this way, make a reference unambiguous.

A pointer-qualified reference contains a dereference. A dereference treats the data object found at the address given by the value of the pointer as an object of the type associated with the pointer.

The forms of a pointer-qualified reference are:

```
name [ ( subscript-list ) ] dereference

dereference [ ( subscript-list ) ]
```

A dereference consists of an "@" character followed by a pointer or a parenthesized pointer formula. That is, the two forms of a dereference are:

```
@ pointer

@ ( pointer-formula )
```

A pointer used in a dereference must be a typed pointer that points to an object of that particular type.

10.3.1.1 Pointers and Ambiguous Names

When two or more tables are declared using a type-name, qualification must be used to make the names of the components unambiguous.

Consider the following declarations:

```
TYPE DIMENSIONS
  TABLE
     BEGIN
     ITEM HEIGHT U;
     ITEM WIDTH U;
     ITEM LENGTH U;
     END
TABLE ROOM DIMENSIONS;
TABLE BOOKCASE DIMENSIONS;
ITEM PTR P DIMENSIONS;
```

The pointer PTR is a typed pointer of type DIMENSIONS. It can be used, therefore, to locate items in a table declared with that type. Assuming the pointer is set to point to the ROOM table, a reference can then be made unambiguously to the item LENGTH in that table using a dereference, as follows:

```
LENGTH @ PTR;
```

The LOC built-in function, which is described in Chapter 12 on "Built-in Functions" is used to obtain a pointer value. For example, to get a pointer to the table ROOM, you can use the LOC function, as follows:

```
PTR = LOC(ROOM);
```

The LOC function returns a typed pointer if its argument is a data object declared using a type-name. In this case, the LOC function returns a pointer of type DIMENSIONS.

Suppose the table declared using the type-name DIMENSIONS is a dimensioned table, as follows:

```
TABLE FACTORY(9) DIMENSIONS;
```

The LOC function can be used to set a pointer to any given entry in that table. For example, suppose you want to reference LENGTH in the first entry of the table FACTORY. You can obtain a pointer of type DIMENSIONS by using the LOC function, as follows:

```
PTR = LOC(FACTORY(0));
```

You can then reference LENGTH as follows:

```
LENGTH @ PTR
```

Suppose a type-name describes a dimensioned table. Consider the following declarations:

```
TYPE SPECIFICATIONS
   TABLE (1:100); DIMENSION;
 TABLE BOXES SPECIFICATIONS;
 ITEM SPECPTR P SPECIFICATIONS;
 ITEM PTR P DIMENSIONS;
```

The table BOXES and the pointer SPECPTR have type SPECIFICATIONS. Each entry in the table has the type DIMENSIONS.

The pointer SPECPTR can be set to point to the table BOXES, as follows:

```
SPECPTR = LOC(BOXES);
```

It can then be used to access the item LENGTH in the first entry of that table, as follows:

```
LENGTH(1) @ SPECPTR
```

Another way of referencing LENGTH in the first entry of the table BOXES is to use PTR, which has associated with it type DIMENSIONS, to point to a particular entry, as follows:

```
PTR = LOC(BOXES(1));
```

The reference, then is:

```
LENGTH @ PTR
```

10.3.1.2 Examples

Consider the following declarations:

```
TYPE DATA
   TABLE;
      ITEM POINT U;

TABLE FIRST DATA;
TABLE SECOND DATA;

ITEM DATAPTR P DATA;
```

The value of the pointer DATAPTR is set by the LOC function, as follows:

```
DATAPTR = LOC(FIRST)
```

The LOC function returns a pointer of type DATA that points to the table FIRST.

Some examples of pointer-qualified references are:

```
@ DATAPTR            --  This pointer-qualified reference
                         references the entire table -- every-
                         thing to which DATAPTR points.

POINT @ DATAPTR      --  This pointer-qualified reference
                         references the item POINT in the
                         table to which DATAPTR points.  In this
                         way, the item POINT in table FIRST is
                         distinguished from the item POINT in
                         the table SECOND.
```

As another example, consider the following declarations

```
TYPE DIMENSIONS
   TABLE (1:15);
      BEGIN
      ITEM LENGTH U;
      ITEM HEIGHT U;
      ITEM WIDTH U;
      END

 TABLE ROOM DIMENSIONS;
 ITEM DIMPTR P DIMENSIONS;
```

The value of the pointer DIMPTR is set as follows:

```
 DIMPTR = LOC(ROOM);
```

The LOC function returns a pointer of type DIMENSIONS that points to the table ROOM.

Some examples of pointer-qualified reference are:

```
@ DIMPTR(13)          --   This pointer-qualified reference
                           references the entire thirteenth entry
                           of the table to which DIMPTR points,
                           (in this case, the table ROOM).

LENGTH(11) @ DIMPTR  --    This pointer-qualified reference
                           references the item LENGTH in the
                           eleventh entry in the table to which
                           DIMPTR points.
```

Chapter 11

FORMULAS

A formula describes the computation of a value. The value of a formula has a type associated with it.

This chapter begins with some general facts about formulas. After that, the remaining sections describe the formulas for each type class: integer, float, fixed, bit, character, status, pointer, and table. For each of these, rules are given for determining the value of a formula and the details of its type. Then, a discussion of compile-time-constant formulas is given.

11.1 FORMULA STRUCTURE

A formula is either a single operand or a combination of operators and operands. A formula has one of the following forms:

```
left-operand  infix-operator  right-operand

prefix-operator  right-operand

operand
```

The type of a formula is determined by the types of its operands. The type classes of the operands of a formula must be the same.

Some examples of formulas are:

```
ALPHA                          ALPHA + 1
FLAG OR STATBIT                NOT MASK
LOC(BOLTS)                     SIZE(INDEX) < BITSINWORD
(ALPHA+1) / (BETA * GAMMA)
```

The first formula is the operand ALPHA. The second formula is the sum of two integer operands. The next two formulas are bit formulas. The next is a function call. The next is a relational expression. The last of these examples is a formula whose main operation is division (as indicated by "/") and whose operands are, themselves, parenthesized formulas. By means of this "nesting" of formulas, one within another, complicated calculations can be written as a single formula.

11.1.1 Operators and Operator Precedence

The JOVIAL (J73) operators are:

Arithmetic Operators	+ - * / ** MOD
Logical Operators	NOT AND OR XOR EQV
Relational Operators	< > = <= >= <>

The operator "+" or "-" can be used either as an infix operator or as a prefix operator. The operator "NOT" can be used only as a prefix operator. The remaining operators can be used only as infix operators.

The order in which the operators and operands are combined is determined by the precedence of the operators. The precedence of the JOVIAL (J73) operators is given in the following table:

Operators	Precedence
**	5
* / MOD	4
+ -	3
< > = <= >= <>	2
NOT AND OR XOR EQV	1

For example, consider the following formula:

```
PI*RADIUS**2
```

The exponontiation operator has precedence 5, and the times operator has precedence 4. Since the exponentiation operator has the higher precedence, it is evaluated first, and then the result is multiplied by PI. Thus the formula just given is equivalent to the following:

```
PI*(RADIUS**2)
```

The effect of precedence on a formula can always be made explicit by adding parentheses to the formula.

Operator precedence does not specify the order in which operators of the same precedence are evaluated. Within a given level of parentheses, the order of evaluation of operators of equal precedence is not specified unless a !LEFTRIGHT directive is in effect. The compiler, in evaluating operators of equal precedence, must observe the laws of commutivity, associativity, and distributivity. That is, the resulting formula must be algebraically equivalent to the original formula.

For example, consider the following formula:

```
Q1+Q2+Q3
```

The compiler can compute Q2 + Q3 first and then add Q1 to the result or it can compute Q1 + Q3 and then add Q2 or it can use any computation that it algebraically equivalent to the above sum.

As another example, consider the following formula:

```
QE**BETA**2
```

The evaluation of the operators in this case cannot be rearranged because the resulting formula is not equivalent. (QE**BETA)**2 is not equivalent to QE**(BETA**2). This formula, therefore, is evaluated from left to right. It is equivalent to the following parenthesized version:

```
(QE**BETA)**2
```

Operator precedence can be overridden by parentheses. For example, suppose you want to express RADIUS as the product of two other radii R1 and R2. You can write:

```
PI*(R1+R2)**2
```

The parentheses force the addition of R1 and R2 to be performed first. Then the exponentiation is performed. Finally the result in multiplied by PI.

You can, and should, use parentheses when you do not feel the grouping of operands with operators is obvious. For example, consider the following formula:

```
(W13/2)/BASE
```

This formula has the same meaning without the parentheses. But the formula is more readable with the parentheses because most people do not know, without consulting the rules given here, which division is performed first. The use of such "extra" parentheses does not slow down execution of the program.

11.1.2 Operands

Each operand of a formula can be any of the following:

Form	Examples
Literal	28.3 'Message 3'
Implementation Parameter	BITSINWORD
Variable	COUNT LENGTH(I)
Constant	PI
Function Call	FACTORIAL(NN)
(Formula)	((ALPHA+1)**2)
Conversion-operator (formula)	U(2*SPEED) (*B 3*)(SPEED)

Conversion operators are discussed in Chapter 13 on "Conversion". Function calls are discussed in Chapter 15 on "Subroutines".

Each operator imposes certain restrictions on the type and value of its operand. For example, the addition operator cannot be applied to a character-literal, and the division operator cannot have zero as its second operand. These restrictions are given later in this chapter, when the various types of formula are described.

11.1.3 Formula Types

The value of each formula has a type (that is, a type class and attributes).

Formulas are classified according to the type of the value of the formula. Under this classification scheme, the permitted types of formula are:

- Integer Formulas
- Float Formulas
- Fixed Formulas
- Bit Formulas
- Character Formulas
- Status Formulas
- Pointer Formulas
- Table Formulas

The remainder of this chapter describes the types of formula according to this classification.

11.2 INTEGER FORMULAS

An integer formula is a formula whose operands are both of an integer type and whose operator is one of the following:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
MOD	modulus

The type of the result of an integer formula is:

```
S n
```

The size, n, is the multiple of BITSINWORD minus 1 used for the larger operand.

For example, suppose you have the following declarations:

```
ITEM LENGTH U 20;
ITEM HEIGHT U 10;
```

If you combine HEIGHT and LENGTH in a formula, the type of the result is:

```
S n   where  if BITSINWORD is 16, n is 31 (2*16-1)
             if BITSINWORD is 24, n is 23 (1*24-1)
             if BITSINWORD is 32, n is 31 (1*31-1)
                    and so on.
```

More examples are given later in this section.

11.2.1 Integer Addition and Subtraction

For an integer formula with "+" or "-" as an infix operator, the result of the formula is the sum or difference of the operands, respectively.

For an integer formula with "+" or "-" as a prefix operator, the result of the formula is the operand or the negation of the operand, respectively.

11.2.2 Integer Multiplication and Division

For an integer formula with the "*" operator, the result is the product of the operands.

For an integer formula with the "/" operator, the result is computed exactly and then truncated, if necessary. No truncation is required if the quotient is an exact integer. Integer division is always truncated even if both operands are declared with a round attribute. Truncation is performed in a machine-dependent manner, either towards zero or towards minus infinity. If the truncation is towards minus infinity, 2.5 is truncated to 2 and -2.5 is truncated to -3. If the truncation is towards zero, 2.5 is truncated to 2 and -2.5 to -2.

The value of the second operand of "/" must not be zero.

11.2.3 Integer Modulus

For an integer formula with the "MOD" operator, the result is the remainder of the division of the first operand by the second. Suppose the values of the operands are v1 and v2. Then the value of the formula is:

```
v1 - (v1/v2) * v2
```

where "/" is integer division as defined in the previous section. Examples will be given later.

The value of the second operand of the MOD operator must not be zero.

11.2.4 Integer Exponentiation

An <u>integer exponentiation formula</u> is a formula whose operator is "**" and (1) whose operands are of type integer (as required for all integer formulas) and (2) whose right operand has a non-negative value that can be calculated at compile time.

The value of an integer exponentiation formula is the same as the value produced by repeated multiplications. If the second operand is n, the result of the formula is the product obtained by multiplying the first operand by itself (n-1) times. If n is Ø, the result is 1.

11.2.5 Examples

Here are examples of integer formulas. For each example, the data type and value of the result is also given.

Suppose the following declarations apply:

```
ITEM BALANCE S 1Ø = 5;
ITEM CONTRIBUTIONS U 2Ø = 6;
ITEM PROFITS U 15 =1Ø;
ITEM FACTOR F = 2.67;
```

Assuming BITSINWORD is 16 and the items still have their initialized values at the time the formulas are evaluated, the following formulas produce the indicated results:

Formula	Result
BALANCE + PROFITS	Value 15, type S 15.
BALANCE-S(FACTOR)	Value 3, type S 15. The floating item FACTOR is converted to a signed integer by the conversion operator S. The conversion truncates the value of FACTOR to 2.
BALANCE*CONTRIBUTIONS	Value 3Ø, type S 31.
PROFITS / 3	Value 3, type S 15.
PROFITS MOD 3	Value 1, type S 15.

11.3 FLOAT FORMULAS

A float formula is either a formula whose operands are both of type float or a float exponentiation formula. A float exponentiation formula is any formula with the "**" operator that is not an integer exponentation formula, as defined previously.

The operator in a float formula must be one of the following:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

The MOD operator is not defined for float operands.

The type of the result of a float formula is:

F n

The precision, n, of the formula is the precision of the operands. If the precisions of the operands are not the same, then the larger precision is used for the type of the result.

Floating formulas are evaluated in an implementation-dependent manner with respect to how exact results are approximated to the implemented precision. The round-or-truncate attribute associated with variables or constants used as operands does not affect the computation of a floating formula result.

11.3.1 Float Addition and Subtraction

For a float formula with "+" or "-" as infix operator, the result of the formula is the sum or difference of the operands, respectively, rounded or truncated in an implementation-dependent manner.

For a float formula with "+" or "-" as prefix operator, the result of the formula is the operand or the negation of the operand, respectively.

11.3.2 Float Multiplication and Division

For a float formula with "*" or "/" as infix operator, the result of the formula is the product or quotient of the operands, respectively.

The value of the second operand of "/" must not be Ø.

11.3.3 Float Exponentiation

A formula whose operator is "**" is a float exonentiation formula unless it is an integer exponentiation formula; that is, unless (1) both operands are of type integer and (2) the second operand is a non-negative compile-time value.

If an operand of a floating exponentiation is an integer, it is converted automatically to a floating type using default precision and rounding before the computation. In floating exponentiation the left operand must not be negative (because a floating point exponent could be a fraction and a negative number raised to a fractional power may produce a complex value).

The value of the formula is the first operand raised to the power specified by the second operand. The value is calculated by a logarithmic method; that is, by the following expression:

$$e^{v2*\log(v1)}$$

where e is 2.73..., log is the natural logarithm function, and v1 and v2 are the first and second operands, respectively.

11.3.4 Examples

Here are some examples of float formulas. For each example, the data type and the value of the formula is given.

Suppose the following declarations apply:

```
ITEM DISTANCE F 15 = 37.2;
ITEM SPEED F,R 12 = 55.;
ITEM COUNT U = 3;
```

Assuming the items still have their initialized values at the time the formulas are evaluated, the following formulas produce the indicated results:

Formula	Result
2.3*DISTANCE	The result is a floating type with precision 15 and the value 85.56.
DISTANCE/SPEED	The result is a floating type with precision 15 and value .6763.
SPEED**2	The result is a floating type with precision 12 and value 3025.0.
SPEED*F(COUNT)	The result is floating type. The precision is that of the most precise operand. The precision of SPEED is 12 and the precision of F(COUNT) is FLOATPRECISION. The value is 165.0.
COUNT**COUNT	The result is a floating type. COUNT is automatically converted to a floating type with precision FLOATPRECISION. Thus the precision of the result is FLOATPRECISION. The value of the result is 27.0.

11.4 FIXED FORMULAS

A fixed formula is a formula with one operand that is fixed and a remaining operand (for infix operators) that is fixed or integer. Its operator must be one of the following:

+	addition
-	subtraction
*	multiplication
/	division

Exponentiation and the MOD operator are not defined for fixed operands.

11.4.1 Addition and Subtraction

Operands of addition and subtraction must have identical scales. For addition and subtraction the scale of the result is the scale of the operands. The fraction of the result is the maximum of the fractions of the operands and the precision is the maximum of the precision of the operands.

11.4.2 Multiplication

For multiplication, two cases are distinguished, one for the case in which one operand is an integer and the other for the case in which both operands are fixed point.

If one operand of a multiplication is an integer, then the scale, fraction, and precision of the result are the same as those of the fixed point operand.

If both operands are fixed point types, then the scale, fraction, and precision of the result are the sum of the scale, fraction, and precision respectively of the operands. If the scale and precision of the result exceeds MAXFIXEDPRECISION or if the scale does not lie in the range -127 through +127, then an explicit conversion must be applied to the result to yield a valid scale and precision.

11.4.3 Division

For division, there are also two cases, one for the case in which the divisor is an integer and one for the case in which both operands are fixed point types.

If the divisor is an integer, the scale and precision of the result are the scale and precision of the fixed point numerator.

If both the numerator and denominator are fixed point types or if the numerator is an integer and the denominator is a fixed point type, the result must be explicitly converted to the desired (legal) scale and precision.

11.4.4 Examples

Here are some examples of fixed point formulas. For each formula, the data type and value is given.

Suppose you have the following declarations:

```
ITEM TIME A 10,5 = 12.5;
ITEM DELTA A 10,5 = .125;
ITEM DISTANCE A 10,-2 = 325.0;
ITEM COUNT U = 4;
```

Assuming the items still have their initialized values at the time the formulas are evaluated, the following formulas produce the indicated results:

Formula	Result
TIME+DELTA	The sum of TIME plus DELTA is a fixed point type with scale 10, fraction 5, and precision 15. The value is 12.625.
6*TIME	The product of 6 times TIME is a fixed point formula with scale 10, fraction 5, and precision 15. The value is 75.0.
DELTA*TIME	The product of DELTA times TIME is a fixed point formula with scale 20, fraction 10 and precision 30. The value is 1.5625.
DISTANCE*TIME	The product of DISTANCE times TIME is a fixed point formula with scale 20, fraction 3, and precision 23. The value is 4162.5.
TIME/COUNT	The quotient of TIME over COUNT is a fixed point type with scale 10, fraction 5, and precision 15. The value is 3.125.
(* A 10,5 *)(TIME/DELTA)	The quotient of TIME over DELTA is a fixed point type whose value is first computed exactly and then converted to a scale of 10 and a fraction of 5 by the conversion operator (*A 10,5*). Conversion operators are described in Chapter 13 on "Conversion". The value is 100.0.

11.5 BIT FORMULAS

A bit formula consists of bit operands and a bit operator. The bit operators are NOT, AND, OR, XOR, and EQV.

11.5.1 Logical Operators

The logical operator NOT produces a value that is the logical complement of its operand. The operators AND, OR (inclusive or), XOR (exclusive or), and EQV (equivalence) perform their usual logical operation on a bit by bit basis, as follows:

Operand Value		Result Value			
Left	Right	AND	OR	XOR	EQV
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

The operands used with a logical operator must have the type bit. When the number of bits in the two operands is not equal, the smaller operand is padded with zero bits on the left until the two operands are the same size. The result has type bit with size equal to the number of bits in the larger operand.

If a formula contains only one kind of logical operator, it can be written without parentheses. For example:

```
FLAG AND STATBIT AND 1B'111'
```

The formula is evaluated in any order unless a !LEFTRIGHT directive is in effect.

However, if a formula contains more than one kind of logical operator, it must contain parentheses to indicate the order of evaluation. For example:

```
FLAG AND (STATBIT OR 1B'111')   or
(FLAG AND STATBIT) OR 1B'111'
```

The NOT operator can be used as a prefix operator only. The other operators are infix operators.

11.5.1.1 Short Circuiting

If the value of a bit formula containing operands of type B 1 is determined before all the operators are evaluated, the evaluation of the remaining operators is omitted or "short-circuited".

For example, consider the following formula:

```
I < 100 OR COEF(I) <> 0
```

If the relational expression I<100 is computed first and if its value is TRUE, the value of the bit formula is TRUE and the relational expression COEF(I)<>0 is not evaluated.

11.5.2 Examples

For example, suppose you have the following declarations:

```
ITEM FLAG B 3 = 1B'010';
ITEM STATBIT B 5 = 1B'00100';
```

Assuming the items still have their initialized values at the time the formulas are evaluated, the following formulas produce the indicated results:

Bit Formula	Result
FLAG AND STATBIT	The AND of FLAG and STATBIT is a bit type five bits long. The shorter item FLAG is padded with zeroes on the left to produce the bit string 1B'00010'. The value of the formula is 1B'00000'.
FLAG OR 1B'10'	The OR of FLAG and the literal 1B'10' is a bit type three bits long. The literal is padded and the value of the formula is 1B'010'
FLAG OR (STATBIT AND 1B'111')	The AND of STATBIT and the literal produces a five bit string with the value 1B'00100'. The OR of FLAG and this formula is a five bit string with the value 1B'00110'.

11.5.3 Relational Operators

A relational operator compares two operands. The result of applying a relational operator to its operand is a relational expression. A relational expression has type bit with size 1.

The relational operators are:

Operator	Meaning
=	Equals
<	Less than
>	Greater than
<>	Not Equal
<=	Less than or Equal to
>=	Greater than or Equal to

The operands in a relational expression must be both of the same type. They can be integer-formulas, floating-formulas, fixed-formulas, character-formulas, status-formulas, or pointer-formulas.
Integer, floating, and fixed comparisons are made on the basis of the value of the operands. Character comparisons are made on the basis of the collating sequence of the character set for a given implementation. Status comparisons are made on the basis of the representation of the status values. Pointer comparisons are made on a target machine dependent basis.

The equals (=) and not equals (<>) operators can be used with bit operands. But the other relational operators cannot be used as no collating sequence is associated with a bit string.

The type of a relational expression is B 1. The value 1B'1' represents the Boolean literal TRUE and the value 1B'0' the Boolean literal FALSE.

11.5.4 Examples

Here are some examples of relational expressions. For each example, the data type and the value of the formula is given.

Suppose the following declarations apply:

```
ITEM COUNT U = 5;
ITEM TIME U = 12;
ITEM DISTANCE F 15 = 37.2;
ITEM SPEED F,R 12 = 55.;
```

Assuming the items still have their initialized values at the time the formulas are evaluated, the following formulas produce the indicated results:

Formula	Result
COUNT<TIME	The type is B 1. The value is TRUE.
SPEED=DISTANCE	The type is B 1. The value is FALSE.

11.6 CHARACTER FORMULAS

A character formula consists of a variable, constant, literal, or function call of type character. In addition, a character formula can be a parenthesized character formula or a bit formula to which a character conversion operator is applied.

Some examples of character formulas are:

```
'Out of Bounds'

(* C 10 *)(CODE)
```

The sequence (* C 10 *) is a conversion operator. Conversion operators are described in Chapter 13 on "Conversion".

No character operators are defined in JOVIAL (J73).

11.7 STATUS FORMULAS

A status formula consists of a variable, constant, literal, function call, or parenthesized formula of type status or a formula converted to type status by a conversion operator.

Some examples of status formulas are:

V(RED)

V(SUNDAY)

11.8 POINTER FORMULAS

A pointer formula consists of a variable, constant, literal, function call, or parenthesized formula of type pointer or formula converted to type pointer by a conversion operator.

Some examples of pointer formulas are:

LOC(CODETAB)

NULL

11.9 TABLE FORMULAS

A table formula consists of a variable, constant, or parenthesized formula whose type class is table or a formula converted to type table by a conversion operator.

Some examples of table formulas are:

GRID

GRID(3)

SPEC(IX*5)

No table operators are defined in JOVIAL (J73).

11.10 COMPILE-TIME-FORMULAS

A compile-time-formula is a formula whose value is required to be computed at compile time by all JOVIAL (J73) compilers. While values of some other formulas are known at compile-time by some or all compilers, they cannot be used where compile-time-formulas are required.

A formula is a compile-time-formula if its operands are taken from the following list:

- A literal
- A status-constant
- An implementation parameter
- A constant item, except for a constant item of type pointer or an item from a constant table.
- A type conversion, except a REP conversion, provided the value of the formula being converted is a compile-time-formula.
- A formula whose operands are compile-time-formulas.
- One of the following built-in functions, subject to the given restrictions:

Function-Name	Restriction
LBOUND, FIRST, LAST	none
UBOUND	The argument cannot be a table with * dimensions.
NEXT, BIT, BYTE, SHIFTL, SHIFTR, ABS, SGN	The arguments of the functions must be compile-time-formulas.
NWDSEN	Its argument must not contain a reference to a name whose declaration is not completed prior to the point at which the function appears. For example, a table T1 cannot contain an item that is preset to NWDSEN(T1).

BITSIZE } BYTESIZE } WORDSIZE }	Their arguments must be compile-time-formulas. An argument must not be either a block or a table with * dimension. An argument must not contain a reference to a name whose declaration is not completed prior to the point at which the function appears.

Consider the following declarations:

```
CONSTANT ITEM VERSION U = 5;
CONSTANT ITEM FACTOR F = 2.36;
CONSTANT ITEM ALPHABET C 26 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
```

Some compile-time-formulas are:

```
2*5-3
VERSION+1
FACTOR**3
BITSINWORD*8
BYTE(ALPHABET,3,6)
```

Chapter 12

BUILT-IN FUNCTIONS

A built-in function is a function that is predefined as part of the JOVIAL (J73) language. Such functions are called in the same way that user-functions are called, but, unlike user-functions, no definition is necessary.

The JOVIAL (J73) built-in functions provide a way of getting information that would otherwise be inaccessible to the user. For example, they provide information about the physical address or physical representation of a data object, the sign of a data object, or the limits of bounds or status lists.

In addition, two functions BIT and BYTE are supplied. These functions select a substring from a bit or character formula, respectively. These functions can also be used as pseudo-variables on the left-hand-side of an assignment statement or other target contexts to set the values of the selected bits.

The JOVIAL (J73) built-in functions are summarized on the following page.

Function-Name	Purpose
LOC	To find the machine address of a data obect, subroutine, or statement.
NEXT	To obtain the arithmetic sum of a pointer argument and an increment, or to obtain the successor or predecessor of a status argument.
BIT	To select a substring from a bit-formula.
BYTE	To select a substring from a character-formula.
SHIFTL SHIFTR	To shift a bit-formula left or right an indicated number of bits.
ABS	To get the absolute value of a numeric-formula.
SGN	To determine if a numeric-formula is negative, zero, or positive.
BITSIZE	To return the logical size in bits.
BYTESIZE	To return the logical size in bytes.
WORDSIZE	To return the logical size in words.
LBOUND UBOUND	To get the lower or upper bound of a given dimension.
NWDSEN	To get the number of words of storage allocated to each entry in a table.
FIRST LAST	To find the lowest or highest value in the status-list argument.

12.1 THE LOC FUNCTION

The LOC function is used to find the machine address of the word in which its argument is contained.

12.1.1 Function Form

The form of the LOC function is:

```
LOC ( argument )
```

The argument of the LOC function can be a data object name, a statement-name, a procedure-name or a function-name. The LOC function returns a pointer value. If the argument of a LOC function is declared using a type-name, then the LOC function returns a typed pointer for the type given in the declaration. Otherwise, the LOC function returns an untyped pointer.

The LOC function is used most often to obtain a value for a pointer to be used in a pointer-qualified reference.

If the argument of a LOC function is a statement-name, procedure-name, or function-name, the LOC function returns the machine address used to access the designated statement or subroutine. The LOC function cannot be used, however, to get the address of a built-in function.

The LOC of a subroutine whose name appears in an inline-declaration or of a statement name within an inline subroutine is implementation defined.

12.1.2 Examples

Suppose you have the following declarations:

```
TYPE GRID
   TABLE;
      BEGIN
      ITEM XCOORD U;
      ITEM YCOORD U;
      END
TABLE BOARD1(2Ø) GRID;
TYPE DIMENSIONS
   TABLE(1Ø);
      BEGIN
      ITEM LENGTH U;
      ITEM HEIGHT U;
      ITEM WIDTH U;
      END
TABLE ROOM DIMENSIONS;
```

You can obtain the machine address of the first entry in the table BOARD1 by using the following LOC function:

```
LOC(BOARD1(Ø))
```

The LOC function of BOARD1(Ø) returns a value whose type is pointer and whose pointed-to attribute is GRID. You can then use that pointer to reference an item in that entry of BOARD1. For example, you can write:

```
XCOORD @ (LOC(BOARD1(Ø)))
```

You can obtain the address of the table ROOM by using the following LOC function:

```
LOC(ROOM)
```

This LOC function returns the address of the table ROOM. The type of this value is pointer with a pointed-to attribute of DIMENSIONS. You can reference an item in the ROOM table as follows:

```
HEIGHT(I) @ (LOC(ROOM))
```

This pointer-qualified-reference locates the item HEIGHT in the Ith entry of the table ROOM.

12.2 THE NEXT FUNCTION

The NEXT function has two separate purposes, depending on the type of its argument. It can be used either to obtain the arithmetic sum of a pointer argument and an increment or to obtain a successor or predecessor of the value of a status formula.

12.2.1 Function Form

The form of the NEXT function is:

```
NEXT ( argument, increment )
```

The argument can be either a status formula or a pointer. The increment is an integer formula.

12.2.2 Status Value Arguments

If the argument is a status formula, the NEXT function returns a successor of the value of the status argument if the increment is positive, or predecessor if the increment is negative.

For example, suppose you have the following status item:

```
ITEM SPECTRUM STATUS
     ( V(RED), V(ORANGE), V(YELLOW),
       V(GREEN), V(BLUE), V(VIOLET) );
```

Now suppose you set the status variable SPECTRUM to V(YELLOW) and then apply the NEXT function as follows:

```
NEXT(SPECTRUM,1)
```

Since the increment is 1, the NEXT function returns the first successor, the status value V(GREEN).

If you give a negative increment, the NEXT function returns the predecessor indicated by the argument. For example, suppose the value of SPECTRUM is V(YELLOW) and you give the following NEXT function:

```
NEXT(SPECTRUM,-2)
```

Since the increment is -2, the NEXT function returns the second predecessor, the status value V(RED).

The increment must not cause the NEXT function to return a value that is outside the status list. Further, the argument of a NEXT function cannot be a status-constant that belongs to more than one status type, unless it is explicitly disambiguated by a conversion operator.

12.2.3 Pointer Value Arguments

If the argument is a pointer formula, the NEXT function returns the arithmetic sum of the pointer formula and the product of the increment times the implementation parameter LOCSINWORD. The type of the result is a pointer of the same type as that of the argument.

For example, consider the use of the NEXT function in the following program fragment:

```
TYPE FORM
   TABLE (100);
      ITEM CODE U;
TABLE CIPHER FORM;
FOR I:LOC(CIPHER) THEN NEXT(I,5) WHILE CODE@I<> 0;
   ACTION(CODE@I);
```

The for-loop examines every fifth entry of the table for a zero code. If the code is not zero, the procedure ACTION is called.

The value of the pointer-formula and the value of the sum of the pointer value and the increment must lie in the implementation-defined set of valid values for a pointers of the given type. The argument of the NEXT function cannot be the pointer literal NULL.

12.3 THE BIT FUNCTION

The BIT function selects a substring from a bit formula. It can be used as a function or as a pseudo-variable.

12.3.1 Function Form

The form of the bit-function is:

```
BIT ( bit-formula, first-bit, length )
```

First-bit and length are integer formulas. First-bit indicates the bit at which the substring to be extracted starts. Length specifies the number of bits in the substring. Bits in a bit string are numbered from left to right, beginning with zero. Length must be greater than zero. The sum of first-bit and length must not exceed the length of the bit-formula.

The type of the result returned by the BIT function is a bit string with size attribute equal to the size attribute of the bit-formula argument. Zeros are automatically added on the left of the value of the result to produce the correct size.

12.3.2 Examples

Suppose you have the following item declaration:

```
ITEM MASK B 10 = 1B'0100110001';
```

And suppose you apply the following BIT function:

```
BIT(MASK,3,4)
```

The BIT function returns a bit string whose rightmost bits have the value of bits 3 through 6 of the bit item MASK. The type of the result is a bit string of length 10. Assuming the item MASK still contains the preset value before the function call, the value of the function is:

```
1B'0000000110'
```

This result is produced by taking bits 3 through 6 of MASK and then padding on the left with zeros to get a 10-bit string.

Since padding and truncation are automatically applied by the compiler for bit strings, you can assign the BIT function to a string of the appropriate length and get the expected result. For example, suppose you have the following declaration:

```
ITEM SUBMASK B 4;
```

You can write the following statement to assign bits 3 through 6 to SUBMASK:

```
SUBMASK = BIT(MASK,3,4);
```

The result of the BIT function is a ten-bit string as indicated above. The six zeros that were automatically used as padding on the execution of the function are automatically truncated when the result is assigned to SUBMASK. Assuming the item MASK still contains its preset value, the value of SUBMASK after the execution of the above statement is:

```
1B'0110'
```

12.3.3 Pseudo-Variable Form

The BIT function can also be used as a pseudo-variable. It can be given on the left-hand side of an assignment statement or as an output parameter. The first argument of a BIT function used as a pseudo-variable must be a variable and not a formula. The form is:

```
BIT ( bit-variable, first-bit, length )
```

The BIT pseudo-variable designates a specified substring of the bit-variable.

12.3.4 Examples

For example, suppose you want to change only the first bit of MASK. You can write the following assignment statement:

```
BIT(MASK,Ø,1)=1B'1';
```

Assuming that MASK has its initial value before this assignment statement, then its value after the assignment is:

```
1B'11ØØ11ØØØ1'
```

12.4 THE BYTE FUNCTION

The BYTE function selects a substring from a character formula. It can be used as a function or as a pseudo-variable.

12.4.1 Function Form

The form of the BYTE function is:

```
BYTE ( character-formula, first-byte, length )
```

First-byte and length are integer formulas. First-byte indicates the character where the substring to be extracted starts, and length specifies the number of characters in the substring to be extracted. Characters are numbered from left to right, starting at zero. Length must be greater than zero. The sum of first-byte and length must not exceed the number of characters in the character formula.

The type of the result returned by the BYTE function is character with a size attribute equal to that of the character-formula argument. Blanks are automatically added on the right to produce the correct size.

12.4.2 Examples

Suppose you have the following item-declaration:

```
ITEM ALPHABET C 26 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ;
```

And suppose you apply the BYTE function as follows:

```
BYTE(ALPHABET,8,4)
```

The BYTE function returns a character string with size attribute 26. Assuming the item ALPHABET still contains its preset value before the function call, the value of the function is:

```
'IJKL
```

That is, it consists of the character sequence IJKL followed by 22 blanks.

Just as in the case of the BIT function, you can assign the BYTE function to a string of the appropriate length and get the expected result since padding and truncation are automatically applied to character strings.

For example, suppose you have the following declaration:

```
ITEM SUBSET C 4;
```

You can write the following statement to assign characters 8 through 11 to SUBSET:

```
SUBSET = BYTE(ALPHABET,8,4);
```

The result of the BYTE function is a 26 character string as indicated above. The 22 blanks that are automatically added to the selected characters on the execution of the function are automatically truncated when the result is assigned to SUBSET. Assuming ALPHABET still has its preset value, the vaue of SUBSET after the execution of the above statement is:

```
'IJKL'
```

12.4.3 Pseudo-Variable Form

The BYTE function can also be used as a pseudo-variable on the left-hand side of an assignment statement or as an output parameter. The form is:

```
BYTE ( character-variable, first-byte, length )
```

The BYTE function in this case designates a specified substring of the character-variable.

12.4.4 Examples

Suppose you have the following:

```
ITEM CODE C 10;
CODE = 'ABQFGHIAAZ';
```

And suppose you want to change characters 5 through 9 of CODE. You can write the following assignment statement:

```
BYTE(CODE,5,5) = 'ZXXXY';
```

Assuming that CODE has the value shown above before this assignment statement, then its value after the assignment is:

```
'ABQFGZXXXY'
```

12.5 SHIFT FUNCTIONS

The shift functions perform logical shifting of a bit formula. Two shift functions are defined, one for left shifting and one for right shifting.

12.5.1 Function Form

The form of the shift functions is as follows:

```
SHIFTL ( bit-formula , shift-count )

SHIFTR ( bit-formula , shift-count )
```

Execution of shift function shifts the bit-formula specified as the first argument by the number of bits specified by the shift-count. Bits that are vacated as a result of the shift are filled with zeros and bits that are shifted out are lost.

Shift-count is an integer-formula. The value of the shift-count must be less than or equal to the implementation parameter MAXBITS, which is the maximum supported value for a bit string. Further, the shift-count must not be negative. If the shift-count is zero, no shift occurs. If the shift-count is greater than or equal to the size of the bit-formula to be shifted, then the result of the function is a bit-string with all zero bits.

The type of the value returned by the shift functions is the same as the type of the bit-formula given as the first argument.

12.5.2 Examples

Suppose you have the following item declaration:

```
ITEM MASK B 5 = 1B'10101';
```

Suppose you apply the SHIFTL function to shift MASK left 3 bits, as follows:

```
SHIFTL(MASK,3)
```

The SHIFTL function returns a bit string of length 5. Assuming MASK still contains its preset value before the shift, the value of the SHIFTL function is:

```
1B'01000'
```

The first three bits were shifted out and lost. The last two bits were shifted left three positions. The remaining three bit positions were filled with zeros.

12.6 SIGN FUNCTIONS

Two sign functions can be applied to numeric formulas, the ABS function and the SGN function.

12.6.1 Function Form

The forms of the sign functions are:

```
ABS ( numeric-formula )

SGN ( numeric-formula )
```

The ABS function produces a value that is the absolute value of the numeric formula. The SGN function returns a value according to whether the numeric-formula is positive, zero, or negative, as follows:

Value of Formula	Value of SGN Function
> Ø	+1
= Ø	Ø
< Ø	-1

The type of the result produced by the ABS function is the same as the type of the numeric-formula argument. The type of the result produced by the SGN function is a signed one-bit integer (S 1).

12.6.2 Examples

For example, suppose you have the following declarations:

```
ITEM TIME U 5 =2;
ITEM VELOCITY F= 2.356;
ITEM RANGE S 10 = -25;
```

Assuming these items still contain their preset values before the function calls, the sign functions produce the indicated values:

Function Call	Function Value
ABS(RANGE)	25
ABS(TIME)	2
ABS(RANGE/TIME)	12 or 13, depending on implementation
SGN(VELOCITY)	+1
SGN(RANGE)	-1
SGN(TIME/RANGE)	0 or -1, depending on implementation

12.7 SIZE FUNCTIONS

The size functions return the logical size of the argument given. Three size functions are defined. BITSIZE returns the size in bits, BYTESIZE returns the size in bytes, and WORDSIZE returns the size in words.

12.7.1 Function Form

The forms of the size functions are:

BITSIZE (size-argument)

BYTESIZE (size-argument)

WORDSIZE (size-argument)

The values returned by the BYTESIZE and WORDSIZE functions are defined in terms of BITSIZE, as follows:

Function	Value	Condition
BYTESIZE	BITSIZE/BITSINBYTE	BITSIZE MOD BITSINBYTE = Ø
	BITSIZE/BITSINBYTE+1	BITSIZE MOD BITSINBYTE <> Ø
WORDSIZE	BITSIZE/BITSINWORD	BITSIZE MOD BITSINWORD = Ø
	BITSIZE/BITSINWORD+1	BITSIZE MOD BITSINWORD <> Ø

The value returned by the BITSIZE function is defined for each of the data types to which the function can be applied in the following sections.

12.7.2 Numeric Data Types

The bitsize of integer and fixed types is related to the size given in the declaration. The bitsize of a floating item is the number of bits actually occupied by the item. The bitsizes for numeric data types are defined as follows:

Data Type	Bitsize
U integer-size	integer-size
S integer-size	integer-size + 1
F	The number of bits actually occupied by the floating item
A scale, fraction	scale+fraction+1

For example, consider the following declarations:

```
ITEM TIME U 5;
ITEM RANGE S 10;
ITEM POSITION U;

ITEM AZIMUTH F 30;
ITEM VELOCITY F;

ITEM SUBTOTAL A 6,2;
```

The following calls return the following values:

Function Call	Function Value
BITSIZE(RANGE)	11
BITSIZE(POSITION)	BITSINWORD-1
BITSIZE(AZIMUTH)	actual number of bits
BITSIZE(VELOCITY*5)	actual number of bits
BITSIZE(SUBTOTAL)	9

Assuming that BITSINWORD is 16 and BITSINBYTE is 8, the following function calls have the following values:

Function Call	Function Value
BITSIZE(POSITION)	15
BYTESIZE(POSITON)	2
WORDSIZE(POSITION)	1
BITSIZE(AZIMUTH)	actual number of bits

12.7.3 Bit and Character Types

The bitsize of a bit type is the bit-size associated with the item in its declaration. The bitsize of a character type is the char-size associated with the item times BITSINBYTE. That is:

Data Type	Bitsize
B bit-size	bit-size
C char-size	char-size * BITSINBYTE

Suppose you have the following declarations:

```
ITEM MASK B 10;
ITEM FLAG B;
ITEM ADDRESS C 26;
ITEM CODE C;
```

Some examples of the result of the BITSIZE function for bit and character types are:

Function Call	Function Value
BITSIZE(MASK)	10
BITSIZE(FLAG)	1
BITSIZE(MASK AND FLAG)	10
BITSIZE(ADDRESS)	26*8
BITSIZE(CODE)	1*8

12.7.4 Status Types

The bitsize of an item with status type is the status size associated with the item in its declaration. The status size is determined by the number of bits necessary to accomodate the representation. The status-size can also be specified in the type-description, as will be seen in Chapter 19 on "Advanced Topics".

Suppose you have the following declarations:

```
ITEM LETTER STATUS
  (V(A),V(B),V(C),V(D),V(E),V(F),V(G),V(H));
ITEM SWITCH STATUS
  (V(ON),V(OFF));
```

The following calls produce the following values:

Function Call	Function Value
BITSIZE(LETTER)	3
BITSIZE(SWITCH)	1

12.7.5 Pointer Types

The bitsize of an item with pointer type is BITSINPOINTER, the implementation dependent parameter that defines the length of a pointer.

12.7.6 Table Types

The bitsize of a table depends on its structure. Table structure is discussed in Chapter 19, "Advanced Topics". Briefly, a table that is specified to have tight structure is one in which as many entries as possible are packed within a word. If the table is not tightly structured, the bitsize of a table or table entry is the number of bits from the leftmost bit of the first word occupied to the rightmost bit of the last word occupied.

For a tightly structured table, the bitsize of the table is the number of bits from the leftmost bit of the first word to the rightmost bit of the last entry. The bitsize of a table entry is either the number of bits specified in the declaration as the size of the entry or, if no size is specified, the number of bits needed for each entry.

Suppose you have the following declarations:

```
TABLE ATTENDANCE (1:10) T 6;
   ITEM COUNT U 5;
TABLE CONDITION(20) T;
   BEGIN
   ITEM ALERT B;
   ITEM CONTROL B;
   END
TABLE SPECIFICATIONS(99);
   BEGIN
   ITEM LENGTH U 5;
   ITEM WIDTH U 9;
   ITEM HEIGHT U 5;
   END
```

The table ATTENDANCE has tight structure with 6 bits per entry. The table CONDITION has tight structure, but the number of bits per entry is not given. The default entry-size is 2.

The following function calls have the following values:

Function Call	Function Value
BITSIZE(ALERT(I))	1
BITSIZE(CONDITION(I))	2
BITSIZE(ATTENDANCE(I))	6
BITSIZE(SPECIFICATIONS)	3*100*BITSINWORD

12.7.7 Blocks

The bitsize of a block is the number of words in the block times BITSINWORD.

Suppose you have the following block declaration:

```
BLOCK GROUP;
   BEGIN
   ITEM COUNT U;
   ITEM VELOCITY F;
   TABLE TIMES(99);
     ITEM SECONDS U;
   END
```

The block GROUP occupies 102 words. Thus the value of BITSIZE(GROUP) is 102*BITSINWORD.

12.8 BOUNDS FUNCTIONS

The bounds functions obtain the bound of a specified dimension of a given table. Two bounds functions are supplied, one to obtain the lower bound and one for the upper bound.

12.8.1 Function Forms

The forms of the bounds functions are:

```
LBOUND ( argument , dimension-number )

UBOUND ( argument , dimension-number )
```

Argument is a table-name.

The LBOUND function returns the lower bound of dimension-number of argument. The UBOUND function returns the upper bound. The dimensions of a table are numbered from left to right, starting at zero.

Dimension-number is a compile-time-integer-formula. It must be greater than or equal to zero and less than the actual number of dimensions for the specified table.

The type of the function value is either integer or status, depending on the declaration of the given table.

12.8.2 Examples

Suppose you have the following declarations:

```
TABLE DATA (1:1Ø,2:2Ø,3:3Ø);
   ITEM DATAPOINT F;
ITEM SEASON STATUS
   (1Ø V(SPRING), V(SUMMER), V(FALL), V(WINTER);
TABLE WEATHER(88,V(WINTER);
   ITEM RAINFALL U;
```

The following calls return the indicated values:

Function Call	Function Value
LBOUND(DATA,Ø)	1
UBOUND(DATA,Ø)	1Ø
LBOUND(DATA,1)	2
UBOUND(DATA,2)	3Ø
LBOUND(WEATHER,1)	V(SPRING)
UBOUND(WEATHER,1)	V(WINTER)

12.8.3 Asterisk Dimensions

If a bounds function is applied to a table that is a formal parameter declared with an asterisk (*) dimension, the bounds function returns the bounds of the table that is the actual parameter, normalized to begin at zero.

The use of the bounds functions makes the following routine a general routine for any two dimensional table with entry attributes that match those of the formal parameter.

```
PROC CLEAR (:TABNAME );
   BEGIN
   TABLE TABNAME (*,*);
      ITEM TABENT U;
   FOR I:Ø BY 1 WHILE I <= UBOUND(TABNAME,Ø);
      FOR J:Ø  BY 1 WHILE J <= UBOUND(TABNAME,1);
         TABENT(I,J) = Ø;
   END
```

The LBOUND function always returns the value Ø for a table declared with asterisk dimensions. Thus, the value Ø, rather than the LBOUND function is used in this example.

You can clear the following two tables using CLEAR:

```
TABLE GRAPH (1:10,2:20);
   ITEM POINT U;
ITEM SEASON STATUS
   (10 V(SPRING), V(SUMMER), V(WINTER), V(FALL);
TABLE WEATHER(88,V(FALL);
   ITEM RAINFALL U;

CLEAR(GRAPH);
CLEAR(WEATHER);
```

As a result of the execution of calls on CLEAR, all the items in the table GRAPH and all the items in the table WEATHER are set to zero.

12.9 THE NWDSEN FUNCTION

The NWDSEN function returns the number of words of storage allocated to each entry in the table or table type given as an argument.

12.9.1 Function Form

The form of the NWDSEN function is:

```
NWDSEN ( argument )
```

The argument can be either a table-name or a table-type-name.

The return type is a signed integer with default size.

12.9.2 Examples

Suppose you have the following declarations:

```
TYPE PART TABLE;
   BEGIN
   ITEM PARTNUMBER U 5;
   ITEM ONHAND U 10;
   ITEM ONORDER U 10;
   END
TABLE BOLTS PART;
TABLE NUTS(100) PART;
```

A table entry of type PART occupies three words. The following calls on NWDSEN produce the following values:

Function Call	Function Value
NWDSEN(PART)	3
NWDSEN(BOLTS)	3
NWDSEN(NUTS)	3

12.10 INVERSE FUNCTIONS

The inverse functions are used to find the lowest and highest permissable values for their argument.

12.10.1 Function Form

The forms of the inverse functions are:

```
FIRST ( argument )

LAST ( argument )
```

The argument can be either a status formula or a status type-name.

The type of the result returned by an inverse function is the same as the type of the argument.

The FIRST function gives the value of the lowest valued status-constant in the status-list associated with the argument and the LAST function gives the value of the highest valued status-constant in that list.

12.1Ø.2 Examples

Suppose you have the following declarations:

```
ITEM LETTER STATUS
  (V(A),V(B),V(C),V(D),V(E),V(F),V(G),V(H));
ITEM SWITCH STATUS
  (V(ON),V(OFF));
```

The following functions have the following results:

Function Call	Function Value
FIRST(LETTER)	V(A)
LAST(LETTER)	V(H)
FIRST(SWITCH)	V(ON)

Chapter 13

CONVERSION

JOVIAL (J73) requires that if a value with one data type is assigned to a data object with a different data type, the source data type must be converted to the target data type. In some cases, the compiler performs the conversion automatically. In other cases, an explicit conversion operator must be supplied.

The following sections discuss contexts for conversion, type equivalence, automatic conversion, and the conversion operators. Then, each data type is considered separately and the data types that are compatible with and convertible to that data type are discussed.

13.1 CONTEXTS FOR CONVERSION

A context that requires conversion is one in which a target and source data object exist, such as: an assignment statement or a subroutine-call. The type of the source data object, in such cases, must be converted to the type of the target data object.

In an assignment statement, the target data object is given on the left-hand-side of the assignment operator (=) and the source data object on the right. Closely related to assignment statements are loop control clauses and presets.

In a subroutine-call, only parameters that are passed by value or value-result are subject to conversion. In these cases, the formal parameter is the target parameter and the actual parameter is the source parameter on entry to the subroutine and, for value-result parameters, the actual parameter is the target and the formal parameter is the source on exit from the subroutine.

13.2 COMPATIBLE DATA TYPES

Data objects are compatible if their types are equivalent or if the compiler automatically converts the source type to the target type.

A data object is equivalent to another data object only if it agrees in type and attributes. The one exception to this rule is a table, in which the names of the items within the tables need not agree for compatible tables.

A data object is automatically convertible to another data type if the compiler performs the conversion. A necessary but not sufficient condition for automatic conversion is that the type classes agree. The cases in which automatic conversion occurs are given for each data type later in this chapter.

13.3 CONVERTIBLE DATA TYPES

A data object is convertible to anoher data type if a conversion operator can be added to make the type of the source equivalent to the type of the target. Three kinds of conversion operator are provided:

```
(* type-description *)
type-indicator
user-type-name
```

The following sections consider each kind of conversion operator in detail.

13.3.1 Type Descriptions

The first kind of conversion operator is a type-description enclosed in the special conversion brackets '(*' and '*)'. The type-description can give the type-class and attributes. The form is:

```
(* type-description *) ( formula )
```

The forms of the type-description were given in Chapter 6 in connection with item-declarations.

For example, suppose you want to assign the floating item RANGE to a 10-bit signed integer and you want the floating item to be rounded before assignment. You can do this by applying a conversion operator that gives the full type-description enclosed in conversion brackets, as follows:

```
INTRANGE =   (* S,R 10 *)(RANGE);
```

If the value of RANGE is 12.526, the value assigned to INTRANGE is 13.

13.3.2 Type-Indicators

Type-indicators are single letter keywords that are used in type-descriptions:

Type-Indicator	Type
U	Unsigned integer
S	Signed integer
F	Floating
B	Bit
C	Character
P	Pointer

The type-indicator for a fixed type, A, is not present in this list because the scale of a fixed type must be given. In all other cases, the attributes of the type-description have defaults.

The type-indicators can be used as conversion operators without the special conversion brackets. The form is:

```
type-indicator ( formula )
```

When a type-indicator is used, the attributes assumed are the same as those assumed for omitted attributes in a declaration.

For example, suppose you want to assign a floating item RANGE to a signed integer. You can do this by using a type-indicator to convert the source data object RANGE, as follows:

```
FIELDRANGE = S(RANGE);
```

The floating item RANGE is converted to a signed integer with the default size BITSINWORD - 1. RANGE is truncated in a machine-dependent manner before assignment.

If FIELDRANGE is declared to be a signed integer of default size, then RANGE is converted and assigned. If FIELDRANGE is declared to be a signed ten-bit integer, then RANGE is converted first to a signed integer of default size by the type-indicator and then to a ten-bit integer by automatic conversion.

13.3.3 User Type-Names

A user type-name is one that is declared in a type-declaration. A user type-name can be used as an abbreviation for a type-description. Like a type-indicator, it can be used as a conversion operator without the conversion brackets, as follows:

```
type-name ( formula )
```

For example, if you have a type-name declared for a 10-bit rounded signed integer, then you can use that type-name to get the same result as the example in which a bracketed type-description was used.

```
TYPE SF S,R 10;
INTRANGE = SF (RANGE);
```

The type-name SF describes the type and attributes of an item and, when it is applied as a conversion operator, the compiler converts RANGE to a ten-bit, rounded, signed integer.

13.4 CONVERSIONS

The following sections consider, for each data type, the types that are compatible with that type and the types that can be converted to that type.

13.4.1 Conversion to an Integer Type

An integer type is one of the type classes S or U with an associated size attribute. An integer type is compatible with any other integer type. Numeric, bit, and pointer types can be converted to an integer type.

13.4.1.1 Compatible Types

An integer type is equivalent to another integer type if both are either S or U and if their size attributes are equal.

An integer type is automatically converted to any other integer type. For example, suppose you have the following declarations:

```
ITEM CARGO'Q2 U,R 20;
ITEM BOX U,T 10;
```

You can write the following assignments:

```
CARGO'Q2 = BOX;
BOX = CARGO'Q2;
```

In the first case, BOX is automatically converted to a 20-bit integer type. In the second case, CARGO'Q2 is automatically converted to a 10-bit integer type.

If the value of the 20-bit integer CARGØ'Q2 requires more than ten bits and if the implemented precision of BOX is not sufficient to hold the value, then some significant bits are truncated. Suppose BITSINWORD is 16. The implemented precision of BOX is 15 and the implemented precision of CARGO'Q2 is 31. If the value of CARGØ'Q2 requires more than 15 bits, truncation occurs when it is assigned to BOX.

13.4.1.2 Convertible Types

Data objects of the following type can be explicity converted by a user-specified conversion operator to an integer type.

integer
float
fixed
bit
pointer

Numeric Conversion -- An integer, floating, or fixed type is converted with the rounding or truncation that is either given or assumed in the conversion operator. Suppose you have the following declarations:

```
ITEM DISTANCE U 10;
ITEM MEASURE F=112.68;
```

Assuming MEASURE has its preset value, the following assignments produce the following values of DISTANCE:

Assignment	Value of DISTANCE
DISTANCE=(*U 10*)(MEASURE);	112
DISTANCE=(*U,R 10*)(MEASURE);	113

The use of a conversion operator results in the loss of most significant digits only if the conversion is from one implemented precision to another. Suppose, for example, DISTANCE is declared to be a five bit integer. That is, we have the following declarations:

```
ITEM DISTANCE U 5;
ITEM MEASURE F=112.68;
```

Consider the following assignment:

```
DISTANCE = (*U 5*)(MEASURE);
```

The value of MEASURE (112.68) is converted to the implemented precision for a five bit integer. Suppose BITSINWORD is 16. The implemented precision then is 15, which is sufficient to hold the value 112, and the value 112 is assigned to DISTANCE.

Bit Conversion --Conversion of a bit string is legal only if the size of the bit string is less than or equal to the bit-size of the integer type. If the size of the bit string is less than the bit-size of the integer, the string is padded on the left with zeroes. For example, suppose you have the following declarations:

```
ITEM MASK B 3;
```

MASK can be explicitly converted to a five-bit integer as follows:

```
(*U 5*)(MASK)
```

The size of the bit string is 3, so it is padded on the left with two zeros. However, MASK cannot be directly converted to a one or two-bit integer.

Pointer Conversion -- Converting a pointer to an integer type is equivalent to first converting the pointer to type B BITSINPOINTER and then converting the bit string to integer. For example, suppose BITSINPOINTER is 24. The following conversion is legal:

```
(*U 24*)(PTR)
```

However, conversion to an integer type whose size is less than BITSINPOINTER is illegal.

13.4.2 Conversion to a Floating Type

A floating type has the type-class F and a precision attribute. A floating type is compatible with any other floating type of equal or greater precision. Integer, floating, fixed, or bit types can be converted to a floating type.

13.4.2.1 Compatible Types

A floating type is equivalent to another floating type if the precision attributes of both are equal.

A floating type is automatically converted to a floating type of greater precision. For example, suppose you have the following items:

```
ITEM POWER F 30;
ITEM FACTOR F 15;
```

You can assign FACTOR as defined above to POWER but not POWER to FACTOR. That is:

```
POWER = FACTOR;       permitted

FACTOR = POWER;       not permitted
```

A real-literal is automatically converted to a floating-literal when it is used as a preset, assignment-value, operand, actual parameter, or initial-value for a loop in connection with a floating data object. The real-literal takes the type of the target value, even if that entails the loss of precision. For example:

```
CONSTANT ITEM PI F = 3.1415926535;
```

Since no precision is given in this declaration, the precision is given by the implementation parameter FLOATPRECISION. If necessary, the value "3.1415926535" is truncated to fit in the number of bits indicated by FLOATPRECISION.

13.4.2.2 Convertible Types

Data objects of the following types can be converted to a floating type by a user-specified-conversion operator:

integer
fixed
float
bit

A user-specified-conversion operator can also be applied to real-literals to convert them to floating types.

Numeric Conversion -- An integer, fixed, or floating type is converted to a floating type with the rounding or truncation specified in the conversion operator. Rounding and truncation are performed with respect to the implemented precision of the type specified by the conversion.

Bit Conversion -- Conversion of a bit string to a floating type is legal only if the size of the bitstring equals the actual number of bits used to represent the floating type. The actual number of bits can be found by using the BITSIZE built-in function, which is described in Chapter 12.

13.4.3 Conversion to a Fixed Type

A fixed type has a type-class A and scale and fraction attributes.

13.4.3.1 Compatible Types

A fixed type is equivalent to another fixed type data object if the scale and fraction attributes of both are equal.

A fixed type is automatically converted to another fixed type with greater scale and fraction attributes. For example, suppose you have the following items:

```
ITEM HEIGHT A 11,4;
ITEM LATITUDE A 10,2;
ITEM LONGITUDE A 10,3;
```

You can assign either LATITUDE or LONGITUDE to HEIGHT, but you cannot assign LONGITUDE to LATITUDE without applying a conversion operator. That is:

```
HEIGHT = LATITUDE;
```

LATITUDE is automatically converted to a fixed type with scale 11 and fraction 4. The assignment of LONGITUDE to LATITUDE requires a conversion operator, as follows:

```
LATITUDE = (*A 10,2*)LONGITUDE;
```

A real-literal is automatically converted to a fixed-literal when it is used as a preset, assignment-value, operand, actual parameter, or initial-value for a loop in connection with a fixed data object.

13.4.3.2 Convertible Types

A user-specified-conversion-operator for fixed conversion can be applied to data object of type integer, fixed, float, and bit. It can also be applied to real-literals.

Numeric Conversion -- An integer, fixed, or floating type is converted to a fixed type with the rounding or truncation specified in the conversion operator. As in the case for floating types, rounding or truncation is performed with respect to the implemented precision of the type specified by the conversion.

Bit Conversion -- Conversion of a bit string to a fixed type is legal only if the size of the bitstring equals the BITSIZE of the fixed type.

13.4.4 Conversion to a Bit Type

A bit type has a type class B and a size attribute.

13.4.4.1 Compatible Types

A data object of type bit is equivalent to another data object of type bit if the size attributes of both are equal.

A bit type is automatically converted to a bit type with a different size attribute by truncating or adding zeros on the left. For example, suppose you have the following items:

```
ITEM MASK B 3 = 1B'010';
ITEM FLAG B = 1B'1';
```

You can assign MASK to FLAG or FLAG to MASK. In the first case, the value of MASK is truncated on the left to produce the value 1B'0', which is then assigned to FLAG.

13.4.4.2 Convertible Types

A bit conversion can be given for any data object except a block. Two types of bit conversion are defined, a user-specified-bit-conversion and a REP conversion.

13.4.4.3 User-Specified Bit Conversion

A user-specified-bit-conversion to a type B NN takes the rightmost NN bits of the data object's representation. If the data object being converted contains less than NN bits, the object is padded on the left with zeros.

If the object being converted is a table or table entry, all filler bits are included in the string. However, if the data object being converted is a character string, filler bits between bytes and unused bytes following the end of the string are not included.

Suppose you have the following declaration:

```
TABLE COEFFICIENTS (3) T 8 = 4(63);
   ITEM CC U 6;
```

This declaration specifies a tight table. A tight table is one in which as many entries as possible are packed within a word. Tight tables are described in Chapter 19 on "Advanced Topics". The table COEFFICIENTS consists of 6-bit unsigned integers, each of which has the decimal value 63, packed in an 8-bit field. The bit pattern of each item equals:

```
63(decimal) = 77(octal) = 1B'111111'(binary)
```

Assuming BITSINWORD is 16, the table COEFFICIENTS has the following pattern:

```
Ø
----------------
xx111111xx111111   word Ø
----------------
xx111111xx111111   word 1
----------------
```

The character "x" indicates a filler bit. That is, a bit that is not set or used.

Now if you apply the following user-specified-conversions, you get the following results:

Conversion Operator	Value
(*B 6*)	1B'111111'
(*B 8*)	1B'xx111111' where x is a filler bit
(*B 16*)	1B'xx111111xx111111'
(*B 2Ø*)	1B'1111xx111111xx111111'
(*B 36*)	1B'ØØØØxx111111xx111111xx111111xx111111'

13.4.4.4 REP Conversions

A REP-conversion obtains the representation of a data object. It converts a data object to a bit string whose size is the actual number of bits occupied by the object.

The form of the REP conversion is:

REP

Suppose you have the following declaration:

```
ITEM COUNT U 3 = 7;
```

If BITSINWORD is 16, the result of the REP conversion is:

```
REP(COUNT) --> 1B'0000000000000111'
```

A REP-conversion can be appplied to named variables only. However, it cannot be applied to tables with * dimensions or to entries in parallel tables.

A REP-conversion can be used on the left-hand-side of an assignment statement.

13.4.5 Conversion to a Character Type

A character type has the type class C and a size-attribute that indicates the number of bytes occupied by the character string.

13.4.5.1 Compatible Types

A character data object is equivalent to another character data object if the size attributes of both are equal.

A character string is automatically converted to a character string by truncating or adding blanks on the right. For example, suppose you have the following declarations:

```
ITEM BOY  C 6  = "NORMAN";
ITEM GIRL C 5  = "TRACY";
```

If you assign the item BOY to the item GIRL, the value of BOY is truncated on the right and the value "NORMA" is assigned to GIRL.

13.4.5.2 Convertible Types

A user-specified-character-conversion can be applied to data objects of bit or character type.

Conversion of a bit string to a character type is legal only if the size of the bitstring equals the actual number of bits used to represent the character type, excluding filler bits between bytes, which can be found by using the BITSIZE built-in function described in Chapter 12. The number of bits in the character string, including filler bits, can be found by first using a REP-conversion.

Consider the following declaration:

```
TABLE NAMES;
   ITEM FIRSTNAME C 8;
```

If BITSINWORD is 36 and BITSINBYTE is 8, the following functions yield the given results:

Call	Result
BITSIZE(NAMES)	72
BITSIZE(FIRSTNAME)	64
BITSIZE(REP(NAMES))	72
BITSIZE(REP(FIRSTNAME))	72

A character string is converted to type C NN by taking the leftmost NN characters. If the data object to be converted contains fewer than NN characters, the value is padded on the right with blanks.

13.4.6 Conversion to a STATUS Type

A status type has type class STATUS and an attribute consisting of a list of status-constants. It can also have a size and specified representations for its status-constants, as described in Chapter 19 on "Advanced Topics".

13.4.6.1 Compatible Types

A data object of type status is equivalent to another data object of type status if both status lists contain the same status values in the same order. In addition, the status size and the representation of the status-constants must also agree.

A status constant that belongs to more than one status list is automatically interpreted unambiguously in the following contexts:

- When it is the source value of an assignment statement, it takes the type of the target variable.
- When it is an actual parameter, it takes the type of the corresponding formal parameter.
- When it is in a table subscript or used in a preset to specify an index, it takes the type of the corresponding dimension in that table's declaration.
- When it is a loop initial-value, it takes the type of the loop-control variable.
- When it is in an item-preset or table-preset, it takes the type of the item or table item being initialized.
- When it is an operand of a relational operator, it takes the type of the other operand.
- When it is in a case-index-group, it takes the type of the case-selector.
- When it is a lower-bound or upper-bound, it takes the type of the other bound.

For example, suppose you have the following declarations:

```
ITEM COLOR STATUS (V(RED),V(ORANGE),V(YELLOW),V(GREEN),
                   V(BLUE),V(VIOLET));
ITEM CONDITION STATUS ((V(RED),V(YELLOW),V(GREEN));
```

The status constants V(RED), V(YELLOW), and V(GREEN) all appear on both the list for COLOR and the list for CONDITION. However, in any of the contexts given above, any ambiguity is automatically resolved by the compiler. For example, if you write:

```
CONDITION = V(RED)
```

The compiler assumes that the type of V(RED) is the same as the type of CONDITION.

The compiler also performs automatic conversion between status types that are the same except for the size attribute.

13.4.6.2 Convertible Types

A user-specified status-conversion can be applied to a data object of bit or status type.

Conversion of a bit string to a status type is legal only if the size of the bitstring equals the actual number of bits used to represent the status type and the range of values of the bit string is within the range of values for the status type. The actual number of bits can be found by using the BITSIZE built-in function, which is described in Chapter 12.

For example, suppose you have the following declarations:

```
TYPE COLOR STATUS (V(RED),V(ORANGE),V(YELLOW),V(GREEN),
                   V(BLUE),V(VIOLET));
ITEM BITS B 3 = 1B'ØØ1';
ITEM SIXBITS B 6 = 1B'ØØØØØ1';
```

You can convert BITS to the status type COLOR because COLOR requires three bits for its representation. Thus, the size of BITS and its value are both valid for conversion purposes. You can convert SIXBITS to the status type COLOR if you provide a conversion operator, as follows:

```
COLOR((*B 3*)(SIXBITS))
```

A status-conversion for a status type is necessary only when the status constant is given in more than one list and is not used in one of the contexts given above.

For example, suppose you give the status-constant V(GREEN) as an upper-bound in a table declaration. If no lower-bound is given or if the lower-bound is also ambiguous, you must use a conversion operator to indicate the type of the status-constant. You can write it as follows:

```
TABLE DATA((*COLOR*)(GREEN));
   ITEM POINT F;
```

13.4.7 Conversion to a Pointer Type

A pointer type has type class P and an attribute that associates a type-name with the pointer.

13.4.7.1 Compatible Types

A pointer data object is equivalent to another pointer data object only if both data objects are untyped or if both are typed with the same type-name attribute. Type-name attributes are considered the same if the names are identical and they are declared in the same type declaration.

A typed pointer is automatically converted to an untyped pointer. For example, suppose you have the following declarations:

```
ITEM P1 P;
ITEM P2 P SUMMARY;
TYPE SUMMARY TABLE;
   ITEM COUNT U;
```

You can assign the typed pointer P2 to the pointer P1. The compiler automatically converts P2 to an untyped pointer. You cannot, however, assign P1 to P2 without first applying an explicit conversion to P1.

13.4.7.2 Convertible Types

A user-specified-pointer-conversion can be applied to a bit, integer or pointer data object.

Conversion of a bit string to a pointer type is legal only if the size of the bitstring equals the actual number of bits used to represent the pointer type. The actual number of bits can be found by using the BITSIZE built-in function, which is described in Chapter 13.

Converting an integer to a pointer is equivalent to first converting the integer to type B BITSINWORD and then converting the bit string to a pointer.

A pointer can be converted to a pointer of another type by the addition of a user-specified-conversion-operator.

13.4.8 Conversion to a Table Type

A table type has type class TABLE and the following attributes:

structure-specifier
number of dimensions
number of elements in each dimension
number of items in each entry
the type and order of each item
the packing of items

13.4.8.1 Compatible Types

Two tables have equivalent types if:

- Their structure specifiers are the same,
- They have the same number of dimensions,
- They have the same number of elements in each dimension,
- They have the same number of items in each entry,
- The types (including attributes) and the textual order of the items are equivalent,
- The explicit or implied packing-spec on each of the items is the same,
- And, the !ORDER directive is either present or absent in both tables.

The names of the items, as well as the types and bounds of the dimension, need not be the same for the tables to be equivalent.

A table entry is considered to have no dimensions.

A table whose entry contains an item-declaration is not considered equivalent to a table whose entry is declared using an unnamed item-description.

The compiler does not perform any automatic conversion for the table type.

13.4.8.2 Convertible Types

A user-specified table conversion can be applied to a data object of type bit or table.

Conversion of a bit string to a table type is legal only if the size of the bitstring equals the actual number of bits used to represent the table type. The actual number of bits can be found by using the BITSIZE built-in function, which is described in Chapter 12.

A table conversion can be applied to a table object of that type merely to assert its type. A table object cannot be converted to a table object of a different type without first converting it to a bit string.

Chapter 14

STATEMENTS

A statement specifies an action that is taken when a program is executed.

14.1 STATEMENT STRUCTURE

A statement is a simple-statement or a compound-statement. A statement can be preceded by labels.

The following paragraphs describe simple-statements, compound-statements, and labels.

14.1.1 Simple-Statements

Simple-statements perform computations, control program flow, and call procedures. A simple-statement is one of the following:

Assignment-Statement
If-Statement
Case-Statement
Loop-Statement
Exit-Statement
Goto-Statement
Procedure-Call-Statement
Return-Statement
Abort-Statement
Stop-Statement
Null-Statement

Each simple-statement is considered in its own section, later in this chapter.

14.1.2 Compound-Statements

A compound-statement groups a sequence of statements together. The grouped sequence of statements can then be used where a single statement is required. The form of a compound-statement is:

```
BEGIN
statement
...

END
```

In this definition, the character sequence "..." below "statement" means that a sequence of any number of statements can appear between BEGIN and END.

Suppose you want to perform several computations if a particular condition is satisfied. You can write:

```
IF LIGHT = V(RED);
   BEGIN
   COUNT = COUNT + 1;
   FACTOR = 2.3 * PREVIOUS;
   PAYOFF =  ANALYSIS(FACTOR);
   END
```

The entire example just given is an if-statement, and the last five lines are a compound-statement. The execution of the if-statement begins with the evaluation of the condition LIGHT = V(RED). If the condition is true, the three statements in the compound-statement are executed; otherwise they are skipped. If the statements were not grouped in a compound-statement, only the assignment to COUNT would be executed conditionally.

14.1.3 Labels

A label is a name followed by a colon, as follows:

```
name :
```

Any number of labels can be placed immediately before a simple-statement. The form is:

```
[ label ... ] simple-statement
```

The square brackets indicate that the labels are optional.

Labels can also be placed immediately before the BEGIN and/or the END of a compound-statement. The form is:

```
     [ label ... ] BEGIN
                   statement
                   ...

     [ label ... ] END
```

A statement is labelled so that it can be the <u>destination</u> of a control statement. Statement labels are also useful for marking sections of code as reference points for documentation or for run-time debugging purposes.

An example of the use of labels is:

```
          BEGIN
          IF COUNT < 2Ø;
             GOTO L1;

            (other statements appear here)

     L1:  IF SPEED > SMAX;
             GOTO L2;

            (other statements appear here)

     L2:  END
```

In this example, the second if-statement is labelled with the statement-name L1, and the END is labelled with L2. If COUNT is less than 2Ø, then some statements are skipped. If SPEED is greater than SMAX, then the remaining statements in the compound-statement are skipped.

The example of labels just given is valid, but it is not a recommended programming style. JOVIAL (J73) provides better ways -- if-statements, case-statements, and loop-statements -- for directing flow of control.

14.1.4 Null-Statements

A null-statement fulfills the requirement for a statement but does not perform any action. The null-statement has one of the following forms:

```
BEGIN [ label ... ] END
```

An example of the use of the null-statement is given in the section on "The Dangling Else" later in this chapter.

14.2 ASSIGNMENT STATEMENTS

An assignment statement evaluates a formula and assigns the result of the evaluation to one or more variables.

An assignment-statement is simple or multiple. A simple assignment-statement sets a given variable to the value of a given formula. A multiple assignment-statement sets storage for several variables.

14.2.1 Simple Assignment-Statements

The form of a simple assignment-statement is:

```
variable = formula ;
```

The formula is evaluated, then the designated variable is located, and finally the value of the formula is placed in that variable.

The data type of the formula on the right of an assignment must be compatible with the data type of the variable given on the left. The data type of the variable is established by the declaration of the variable. The data type of the formula is determined by the data types of the operands, as described in Chapter 11 on "Formulas".

For example, you can write:

```
FORCE = MASS * ACCELERATION;
```

This statement computes the product of MASS times ACCELERATION and assigns that value to the variable FORCE. The data type of the formula is determined by the data types of MASS and ACCELERATION. If that data type is compatible with the data type declared for FORCE, the assignment is valid.

14.2.2 Multiple Assignment-Statements

The form of a multiple assignment-statement is:

```
variable ,... = formula ;
```

The formula is evaluated first. Then the variables are processed, from left to right. For each variable, the designated variable is located and the value of the formula is placed in that variable.

For example, suppose you want to set three variables to zero. You can write:

```
TIME, DATE, STATUS = 0;
```

The type class of all the variables on the left side of the assignment must be the same. The type of the formula must be compatible with the type of each of the variables given.

For example, suppose you have the following declarations:

```
ITEM HEIGHT U 5;
ITEM LENGTH U 10;
ITEM SIZE U 15;
```

The following assignment is valid:

```
HEIGHT, LENGTH = SIZE;
```

The type class of HEIGHT and LENGTH is the same (U). The type of the formula is U 15, which is compatible with the type of HEIGHT (U 5) and with the type of LENGTH (U 10).

The order in which assignments are performed is sometimes significant. Suppose, for example, you write the following assignment statement

```
INDEX, PARTS(INDEX) = 5;
```

The leftmost variable INDEX is processed first and assigned the value 5. The next variable PARTS(INDEX) is processed next. Since the value of INDEX has already been changed to 5, PARTS(5) is assigned the value 5.

However, if you give the variables in a different order, the result of the assignment could be different. Suppose the value of INDEX is 1 and you write:

```
PARTS(INDEX), INDEX = 5;
```

This statement assigns the value 5 first to PARTS(INDEX). Since INDEX is 1, PARTS(1) receives the value 5. Then the statement assigns the value 5 to INDEX.

14.3 <u>IF-STATEMENTS</u>

An if-statement controls the flow of a program. The simplest form of if-statement executes a statement when a given <u>condition</u> is true. The form of this if-statement is:

```
IF test ;

   true-alternative
```

Test is a Boolean formula. If the value of test is TRUE, the true-alternative is executed; otherwise, the true-alternative is skipped.

For example, suppose you want to call an error routine if the value of a counter exceeds a specified threshold. You can write the following if-statement:

```
IF COUNT > THRESHOLD;
   ERROR(11);
```

If the value of COUNT is greater than the value of THRESHOLD, the value of test is TRUE and the true-alternative, which invokes the procedure ERROR, is executed. If COUNT is less than or equal to THRESHOLD, the value of test is FALSE and control passes to the next statement.

A second form of the if-statement executes either of two given statements, depending on the value of the test. The form is:

```
IF test ;

   true-alternative

ELSE

   false-alternative
```

If the value of test is TRUE, then the true-alternative is executed; otherwise, the false-alternative is executed.

The following statement is an example of the second form of if-statement:

```
IF INDIC = V(RED);
   COUNT1=COUNT1+1;
ELSE
   COUNT2=COUNT2+1;
```

This statement increments COUNT1 if INDIC = V(RED) and COUNT2 otherwise.

14.3.1 Compound Alternatives

True-alternative and false-alternative are each a single statement. A compound-statement can be used to include more than one statement in a true-alternative or false-alternative.

An example of a compound-statement in an if-statement appears earlier in this chapter, under "Compound-Statements". Another example is:

```
IF INDIC = V(RED);
   COUNT1 = COUNT1 + 1;
ELSE
   BEGIN
   COUNT2 = COUNT2 + 1;
   MASK = FLAGS AND MONITOR;
   END
```

In this example, true-alternative is a simple-statement, but false-alternative is a compound-statement that groups two simple-statements together. When the value of test is FALSE, the statement not only increments COUNT2 but also sets the variable MASK.

14.3.2 Nested If-Statements

If-statements can be nested, one inside another, to perform complex tests. For example, suppose you want to call one of four procedures based on the value of the status variable COND and the counter COUNT, as follows:

```
IF COND = V(RED);
   IF COUNT < TX;
      CASE1(TMAX,COUNT);
   ELSE
      CASE2(TMAX,COUNT);
ELSE
   IF COUNT < TX;
      CASE3(JMAX,THRESHOLD);
   ELSE
      CASE4(JMAX,THRESHOLD);
```

If COND is V(RED) and COUNT is less than TX, the procedure CASE1 is called. If COND is V(RED) and COUNT is not less than TX, the procedure CASE2 is called. If COND is not V(RED) and COUNT is less than TX, the procedure CASE3 is called. If COND is not V(RED) and COUNT is not less than TX, the procedure CASE4 is called.

The behavior of this if-statement is diagrammed in the following table:

COND = V(RED)		COND <> V(RED)	
COUNT < TX	COUNT >= TX	COUNT < TX	COUNT >= TX
CASE1	CASE2	CASE3	CASE4

14.3.3 The Dangling ELSE

In a nested if-statement, an ELSE clause that could be part of several if-statements is sometimes called a "dangling ELSE". In JOVIAL, such an ELSE clause is associated with the closest of the statements of which it could be a part. As an example of a dangling ELSE, consider the following:

```
IF COND = V(RED);
   IF FF = V(SET);
      ACTION = 1;
   ELSE
      ACTION = 2;
```

The ELSE clause is associated with the closer if-statement, which contains the test FF = V(SET). The action taken by this if-statement is shown in the following table:

COND = V(RED)		COND <> V(RED)
FF = V(SET)	FF <> V(SET)	(No action)
ACTION = 1	ACTION = 2	

If you want to associate the dangling ELSE with the test COND = V(RED) rather than with the test FF = V(SET), you can use a compound-statement as follows:

```
IF COND = V(RED);
   BEGIN
   IF FF = V(SET);
      ACTION = 1;
   END
ELSE
   ACTION = 2;
```

Alternatively, you can use a null-statement as the false-alternative for the test FF = V(SET), as follows:

```
IF COND = V(RED);
   IF FF = V(SET);
      ACTION = 1;
   ELSE;
ELSE
   ACTION = 2;
```

The action taken by these statements is equivalent. It is shown in the following table:

<table>
<tr><td colspan="2">COND = V(RED)</td><td>COND <> V(RED)</td></tr>
<tr><td>FF = V(SET)</td><td>FF <> V(SET)</td><td rowspan="2">ACTION = 2</td></tr>
<tr><td>ACTION = 1</td><td>(No action)</td></tr>
</table>

Note that the indentation of the ELSE clause in the first example of this section is differenet from the indentation in th second and third examples. The purpose of the indentation is to make the intent of the code clearer to the reader. Indentation has no effect on the syntax analysis of a JOVIAL (J73) program.

14.3.4 <u>Compile-Time-Constant Tests</u>

If test in an if-statement is a compile-time-formula, the compiler evaluates test and reduces the if-statement to a single statement. The compiler generates object code for the selected alternative, but not for the test or the other alternative.

The compiler examines the unselected alternative at compile-time. Thus, it must be syntactically correct and directives in the unselected alternative are processed. However, since no object code is generated for the unselected alternative, labels in that alternative are not defined when the program is executed and cannot be used either in goto-statements or as actual parameters outside the alternative.

The same label cannot be used in both the alternatives even though only one alternative is selected. All labels in a scope must be unique.

14.4 CASE-STATEMENTS

A case-statement provides for the execution of one or more of a number of statements based on the value of the case-selector. The case-statement has the following form:

```
     CASE  case-selector ;

        BEGIN

        [ default-option ]

        case-option    ...

        END
```

The square brackets indicate that the default-option can be omitted. The sequence "..." indicates that one or more case-options can be given.

Case-option has the form:

```
     (  case-index ,... ) : statement
```

The value of case-selector determines the option that is executed. After the execution of that option, control passes to the statement after the case-statement (that is, following the END) unless a FALLTHRU clause is given. The FALLTHRU clause is explained a little later in this section.

Case-selector can be an integer, bit, character, or status formula. The case-indexes designate the statement to be performed for a particular value of the case-selector.

If the value of the case-selector does not lie in the specified ranges, then the statement associated with the DEFAULT selection, if present, is selected. The default-option has the form:

```
( DEFAULT ) : statement
```

For example, suppose you want to perform a calculation based on whether a number in the range 1 through 2Ø is prime or non-prime. You can write the following case-statement:

```
CASE FACTOR;
   BEGIN
   DEFAULT: ERROR(FACTOR);
   (1,3,5,7,11,13,17,19): PRIME;
   (2,4,6,8,9,1Ø,12,14,15,16,18,2Ø): NONPRIME;
   END
```

If the value of FACTOR is a prime number in the range from 1 through 2Ø, the procedure PRIME is called. If the value of FACTOR is in that range but is not a prime, NONPRIME is called. If the value of FACTOR is outside the specified range, an error procedure is called.

If default-option is given, it must be the first option in the case-statement. If it is not given, then it is an error if a run-time value of the case-selector has a value for which there is no case-option. In this circumstance, the effect of the case-statement is undefined.

The type of case-index must be compatible with the type of case-selector. The case-indexes must be distinct, so that a value of the case-selector unambigously selects a case-index and an associated case-option.

Each case-index must be known at compile time. It can be any integer, bit, character, or status compile-time-formula.

14.4.1 Bound Pairs

A case index can be a bound-pair. The form of a bound-pair is:

```
first-case : last-case
```

For example, consider the following case-statement:

```
CASE  ACTIONSELECT;
  BEGIN
  (DEFAULT):                 ACTION(1);
  (1:5,101,103,105:107)  ACTION(2);
  (6:10):                    ACTION(3);
  (11:100):                  ACTION(4);
  END
```

If the value of the case-selector ACTIONSELECT is between 1 and 5, equal to 101 or 103, or between 105 and 107, then the procedure ACTION is called with the parameter 2. If the value of ACTIONSELECT is between 6 and 10, then ACTION is called with parameter 3, and so on. If the value is less than 1 or greater than 100, then the statement associated with the default option is executed and, in this case, ACTION is called with the parameter 1.

14.4.2 The FALLTHRU Clause

An option in a case-statement can also have a FALLTHRU clause. The FALLTHRU clause follows the statement in an option, as follows:

```
CASE  case-selector ;

   BEGIN

   [ ( DEFAULT ) :          statement  [ FALLTHRU ] ]

   ( case-index, ... ) :  statement  [ FALLTHRU ]
   ...

   END
```

The FALLTHRU clause can be present on any or all of the options in a case-statement.

If, when a case-statement is executed, a FALLTHRU clause is present on the selected option, then after the execution of the statement in that option, the statement in the next option is executed. Then, if that option has a FALLTHRU clause, the statement in the next option is executed after the execution of the statement in the current option is complete. This "falling through" continues until the case-statement is terminated either by an option that does not have a FALLTHRU clause or by the end of the case statement.

Suppose you want to use the value of PROFIT to determine which of a set of procedures is executed. You can write the following CASE statement:

```
CASE PROFIT;
   BEGIN
   ( DEFAULT ): ERROR(21);
   (1000:9999): DIVIDEND(PROFIT); FALLTHRU
   (500:999):   BALANCE(PROFIT);
   (100:499):   REEVALUATE;
   (0:99):      CLOSEOUT;
   END
```

If the value of the case-selector PROFIT is between 1000 and 9999, the option for that case-index is executed. The procedure DIVIDEND is called, and then, since that option has a FALLTHRU clause, the procedure-call BALANCE, which is the statement in the next option, is executed. If the value of PROFIT is between 500 and 999, the procedure BALANCE is called. If PROFIT is between 100 and 499, REEVALUTE is called. If PROFIT is between 0 and 99, CLOSEOUT is called. Any other value of PROFIT causes the default-option to be selected and the procedure ERROR to be called.

14.4.3 Compile-Time-Constant Conditions

If the case-selector in a case-statement is a compile-time-formula, as defined in Chapter 11, the compiler evaluates it and reduces the case-statement to the appropriate statements. In this case, the compiler generates object code for the selected option and, if the selected option contains a FALLTHRU clause, for all statements selected by the FALLTHRU sequence of the case-statement.

The compiler examines the unselected options at compile-time. Thus, they must be syntactically correct. Directives in the unselected options are processed. No object code, however, is generated for the unselected options, and labels in those options cannot be used either in goto-statements or as actual parameters outside those options. The labels in an unselected option do not exist at run time. Even so, all labels in a case-statement must be unique.

14.5 LOOP-STATEMENTS

A loop-statement provides for the iterative execution of a statement. A loop-statement is a while-loop or a for-loop.

14.5.1 While-Loops

A while-loop specifies a condition which, if true, calls for the execution of the statement within the loop. As long as that condition is true, the statement is executed. If the condition is false, control passes to the next statement. The form of a while-loop is:

```
WHILE  condition ; statement
```

Condition is a Boolean formula. The statement in a while-loop is executed repeatedly as long as the value of the condition is TRUE. Each repetition begins with an evaluation of the condition and continues, if the value of condition is TRUE, with an execution of the statement.

For example, suppose you want to execute a case-statement as long as the value supplied by each execution lies within a given range. You can write the following while-loop:

```
WHILE READY;
   CASE INDEX;
      BEGIN
      (DEFAULT):    READY = FALSE;
      (1:10):       INDEX = LEVEL1(INDEX);
      (11:20):      INDEX = LEVEL2(INDEX);
      (21:100):     INDEX = PEAK(INDEX);
      (101:200):    INDEX = SUPER(INDEX);
      END
```

In this example, the case-options each call a function that performs some computations and returns a new value for INDEX. If the value of INDEX is not within the specified ranges for case-indexes, the default-option sets READY to FALSE. Until that circumstance arises, the case-statement is executed repeatedly.

14.5.2 For-Loops

The for-loop includes a mechanism for setting and changing a variable, the loop-control. One form is:

```
FOR  loop-control : initial-value ;

    statement
```

The initial-value is evaluated and assigned to loop-control and then the statement is executed repeatedly. That is, the execution is as follows:

1. Evaluate initial-value and assign its value to loop-control.

2. Execute the statement.

3. Return to Step 2.

In this case, some condition within the statement must transfer outside the loop to terminate the for-loop.

The for-loop can also include a WHILE phrase, as follows:

```
FOR loop-control : initial-value [ WHILE condition ] ;

        statement
```

If a WHILE phrase is given, the condition in the WHILE phrase governs the number of times the for-loop is executed. The execution of the statement is as follows:

1. Evaluate initial-value and assign its value to loop-control.

2. Evaluate the condition in the WHILE phrase. If the value of the condition is TRUE, continue to step 3. If the value of the condition is FALSE, terminate the for-loop.

3. Execute the statement.

4. Return to step 2.

The for-loop can also include a clause that changes the value of the loop-control in either of two ways, by incrementation (or decrementation) or by repeated assignment.

14.5.2.1 Incremented For-Loops

An incremented for-loop has the following form:

```
FOR loop-control
    : initial-value BY increment [ WHILE condition ] ;

    statement
```

The square brackets indicate that the WHILE phrase is optional.

A BY clause indicates that the value of increment is to be added to loop-control.

If a WHILE phrase is not given, initial-value is evaluated and its value assigned to loop-control, the statement is executed, and then the value of loop-control is changed. This process continues until some condition within the statement transfers outside the loop to terminate the for-loop.

If a WHILE phrase is given, the condition in the WHILE phrase governs the number of times the for-loop is executed.

The while phrase can also be given before the BY clause, as follows:

```
FOR loop-control
      : initial-value [ WHILE condition ] BY increment

      statement
```

The execution of the statement is as follows:

1. Evaluate initial-value and assign its value to loop-control.

2. If a WHILE phrase is present, evaluate the condition in the WHILE phrase. If the value of the condition is TRUE, continue to Step 3. Otherwise, terminate the for-loop.

3. Execute the statement.

4. Evaluate increment and add its vaue to the value of loop-control.

5. Return to step 2.

Suppose, for example, you want to exchange the items of two tables. Each table has 25 entries. You can write:

```
FOR IX:Ø BY 1 WHILE IX<25;
   BEGIN
   TEMP = DAY(IX);
   DAY(IX) = NIGHT(IX);
   NIGHT(IX) = TEMP;
   END
```

This statement sets the loop-control item IX to Ø and evaluates the condition in the WHILE phrase. Since the value of the loop-control IX is less than 25, the statement exchanging DAY(Ø) and NIGHT(Ø) is executed. Then, IX is incremented by 1 and the WHILE phrase condition is evaluated again. The value of the loop-control is now 1, which is less than 25, and the statement is executed again, exchanging DAY(1) and NIGHT(1). This process continues until the loop-control is 25 and the condition is false. Control then passes to the next statement.

14.5.2.2 Repeated Assigment Loops

The form of a repeated assignment loop is:

```
FOR loop-control
    : initial-value THEN formula [ WHILE condition ] ;

    statement
```

The THEN clause indicates that the value of the formula is to be <u>assigned</u> to the loop-control on each iteration of the loop.

If a WHILE phrase is given, the condition in the WHILE phrase governs the number of times the for-loop is executed. The WHILE phrase can be given before the THEN clause in a repeated assignment loop just as it can be given before the TO clause in an incremented loop.

The execution of a repeated-assignment loop is as follows:

1. Evaluate the initial-value and assign it to the loop-control.

2. If a WHILE phrase is present, evaluate the condition in the WHILE phrase. If the value of the condition is TRUE, continue to Step 3. Otherwise, terminate the for-loop.

3. Execute the statement.

4. Evaluate the formula in the THEN clause and assign it to loop-control.

5. Return to step 2.

As an example of the THEN clause, suppose you have the entries of a table linked in a list. Each entry contains two items. The first item, VALUE, contains a value and the second item, LINK, contains an index to the next entry in the list. The beginning of the list is given in the item LISTHEAD and the end of the list is indicated by a Ø link. You can process each item in the list by following the links to the end as follows:

```
FOR IX:LISTHEAD  THEN LINK(IX) WHILE IX <> Ø;
   PROCESS(VALUE(IX));
```

This statement sets the loop-control IX to the beginning of the list. If that value is zero, then the list is null. That is, it contains no entries. If it is not zero, then the body of the loop, which invokes the procedure PROCESS, is executed. Then the link LINK(IX) is assigned to IX. If IX is not zero, the statement is executed again. This process continues until the end of the list is reached.

14.5.3 Loop-Control

The loop-control in a for-loop with a BY or THEN clause receives a new value for each iteration of the loop.

Loop-control can be an item-name or a single letter. A single letter loop-control is implicitly declared for the loop statement. If loop-control is a declared item, the formulas given to set and change it must be compatible in type with loop-control. If loop-control is a single letter, the values given in the BY or THEN clauses must be compatible with the initial-value. In either case, the formula given in a BY clause must have data type and value such that it can be added to the loop-control.

The value of loop-control should not be altered in a loop-statement. The compiler does not allow the value of a single letter loop-control to be changed in a for-loop. It allows the value of a declared item loop-control to be changed, but it issues a warning message in that case.

If the value of loop-control is not needed before or after the execution of the for-loop, a single letter loop-control is convenient. It does not require declaration and its scope is the for-loop statement itself. Thus no conflict with other loop-controls can exist.

For example, suppose you want to perform a computation for all items, COUNT, of a table with 100 entries. You can write:

```
FOR I:1 BY 1 WHILE I <= 100;
   COMPUTE(COUNT(I));
```

This loop calls the procedure COMPUTE for each entry in the table.

If the value of loop-control is needed for use after the execution of the loop statement, a declared item-name rather than a single letter loop-control must be used.

Suppose you want to sum all the items of the table but you also want to terminate the loop if the sum exceeds a given threshold. If you want to know the index of the item that caused the sum to exceed the threshold after the loop is terminated, you must use a declared item for loop-control, as follows:

```
FOR INDEX:1 BY 1 WHILE INDEX <= 100;
   BEGIN
   SUM = SUM + COUNT(INDEX);
   IF SUM > TMAX;
      GOTO OVERFLOW;
   END
```

If SUM exceeds the threshold, control is transferred to OVERFLOW and the value of INDEX can be used to determine the item at which the overflow condition occurred.

14.5.4 Labels within For-Loops

A label within the body of a for-loop cannot be used outside the for-loop. That is, control cannot be sent into a for-loop. The body of a for-loop can be executed only by executing the for-loop statement. A GOTO statement within the for-loop, however, can transfer control to a labelled statement within the for-loop.

14.6 EXIT-STATEMENTS

An exit-statement is used to terminate a loop at the point within the loop where the exit-statement is executed.

An exit-statement terminates the execution of the immediately enclosing loop-statement. The form of the exit-statement is:

```
EXIT ;
```

The effect of an exit-statement is the same as the effect of a GOTO statement that transfers control out of the controlled-statement to the point following the loop-statement.

For example, suppose you are processing a table of 1ØØØ entries, each of which contains three items, and you want to terminate if the value of one of the items is zero. You can include a test in the WHILE phrase as follows:

```
FOR I:Ø BY 1 WHILE HEIGHT(I) <> Ø AND I <= 999;
   BEGIN
   PROCESS(LENGTH(I));
   PROCESS(HEIGHT(I)*WIDTH(I));
   END
```

This statement performs the computations if HEIGHT(I) is not zero. If HEIGHT(I) is zero, the for-loop is terminated.

However, suppose you want to call the procedure PROCESS for the item LENGTH(I) before terminating if the value of HEIGHT(I) is zero. Then you can use the exit-statement, as follows:

```
FOR I:Ø BY 1 WHILE I <= 999;
   BEGIN
   PROCESS(LENGTH(I));
   IF HEIGHT(I) = Ø;
      EXIT;
   PROCESS(HEIGHT(I)*WIDTH(I));
   END
```

This statement performs the computations if HEIGHT(I) is not zero. If HEIGHT(I) is zero, it invokes PROCESS for LENGTH(I) and then terminates.

14.7 GOTO-STATEMENTS

A goto-statement transfers control to the statement labelled by the given statement-name. The form is:

```
GOTO statement-name ;
```

Statement-name must be known in the scope in which the goto-statement appears. It must not be the label of a statement within the controlled-statement of a loop-statement, unless the goto-statement is also within that same controlled-statement or within a nested controlled-statement. Further, it must not be the label of a statement that is in an enclosing subroutine or in another module.

A goto-statement should not be used for the cases in which JOVIAL supplies another statement for the transfer of control. The exit-statement, for example, terminates a for-loop statement and the return- and abort-statements terminate a subroutine. Cases exist, however, in which the goto-statement can be effectively used.

For example, suppose you are summing the items of a table with two dimensions and you want to terminate the summation process if the sum exceeds a specified threshold. Since the table has two dimensions, the summation process requires a nested for-loop. You can use a GOTO statement to terminate the execution of both for-loops, as follows:

```
FOR I:1 BY 1 WHILE I < 100;
   FOR J:1 BY 1 WHILE J < 100;
      BEGIN
      SUM = SUM + COUNT(I,J);
      IF SUM > TMAX;
         GOTO OVERFLOW;
      END
```

14.8 PROCEDURE-CALL-STATEMENTS

A procedure-call-statement invokes the named procedure and associates the actual parameters of the call with the formal parameters of the definition. The form is:

```
procedure-name [ ( actual-list ) ]
                                  [ ABORT statement-name ] ;
```

Both the parenthesized parameter list and the ABORT phrase are optional. If a procedure-definition does not declare formal parameters then the call does not include any actual parameters. The ABORT phrase is used to provide a statement-name to be used in connection with any ABORT statement within the procedure or procedures called by the procedure.

The procedure-call statement is described in detail in Chapter 15 on "Subroutines", in which subroutine definitions and subroutine calls are considered together.

14.9 RETURN-STATEMENTS

A return-statement effects a normal return from a subroutine. The form is:

```
RETURN ;
```

The return-statement sets any parameters for normal subroutine termination and then transfers control to the point following the invocation of the subroutine.

The return-statement is described and illustrated in Chapter 15 on "Subroutines".

14.10 ABORT-STATEMENTS

An abort-statement effects an abnormal return from a procedure. The form is:

```
ABORT ;
```

When an abort-statement is executed, control passes to the statement-name given in the most recently executed, currently active procedure-call-statement that has an abort phrase. If no currently-active procedure-call-statement has an abort phrase, then the effect of an abort-statement is the same as that of a STOP statement.

When an abort-statement is executed, all intervening subroutine invocations are terminated and the parameters of these subroutines are not set (as they would be if normal termination of the subroutine occurred).

The abort-statement is further described and illustrated in Chapter 15 on "Subroutines".

14.11 STOP-STATEMENTS

A stop-statement causes execution of the program to terminate. The form is:

```
STOP [ stop-code ] ;
```

If the stop-code is given, it is an integer formula whose value is supplied to the environment in which the JOVIAL program is running. The meaning of the value of the stop-code is implementation-dependent. The range of legal values for stop-code is defined by the implementation parameters MINSTOP through MAXSTOP.

If a stop-statement is executed within a subroutine, the parameters of any active subroutine are not set as they would be on normal termination of the subroutine.

Chapter 15

SUBROUTINES

A subroutine is an algorithm that can be executed from more than one place in a program. A subroutine can be either a procedure or a function. A procedure is executed by a procedure-call statement. A function returns a value and is executed within a formula by a function-call.

The following sections describe the definition and use of procedures and functions.

15.1 PROCEDURES

A procedure describes a self-contained portion of a program. A procedure can interact with its environment through its parameters or global data. A procedure-definition defines the name, attributes, and logic of a procedure. A procedure-call invokes that logic and supplies actual parameters to be used in the execution of the procedure's statements.

15.1.1 Procedure-Definitions

A JOVIAL (J73) procedure-definition gives the procedure name, declarations for all formal parameters and local data, the executable statements of the procedure and the definitions of any subroutines local to the procedure.

A procedure-definition has the following form:

```
PROC procedure-name

        [ use-attribute ] [ ( formal-list ) ] ;

            procedure-body
```

The square brackets indicate that use-attribute and the parenthesized list of formal parameters can be omitted. Use-attribute indicates whether the subroutine is recursive or reentrant. The compiler uses this attribute to allocate data within the procedure properly. Use-attribute and the parameters are described in detail later in this chapter.

Procedure-body contains the declarations of any parameters, declarations of any local data, the statements of the procedure, and definitions of any local subroutines used in the procedure.

Procedure-body can be simple or compound. In either case, it must contain at least one executable statement.

15.1.2 Simple Procedure-Bodies

The simplest form of a procedure-body contains only an executable statement, as follows:

```
statement
```

Only a procedure without parameters can have a simple procedure-body, because parameters must be declared. If a procedure has parameters, it must have a compound procedure-body.

As an example of the simple form of a procedure-body, suppose you want to write a procedure TABMULT that multiplies each item in one table by the corresponding item in another table and saves the product in a third table. The tables to be multiplied are declared as follows:

```
TABLE PHASE1(1:1ØØØ);
   ITEM COUNTP1 F;
TABLE PHASE2(1:1ØØØ);
   ITEM COUNTP2 F;
TABLE PHASES(1:1ØØØ);
   ITEM RESULTS F;
```

You can write the procedure TABMULT as follows:

```
PROC TABMULT;
  FOR I:1 BY 1 WHILE I <= 1ØØØ;
     RESULTS(I) = COUNTP1(I)*COUNTP2(I);
```

TABMULT is a very specialized routine that only works for the given tables. A more general routine for this kind of table manipulation is described later in this chapter.

15.1.3 Compound Procedure-Bodies

The compound form of a procedure-body can contain declarations, statements, and definitions of subroutines used in the computation, as follows:

```
BEGIN

[ declaration  ...  ]

statement ...

[ subroutine-definition  ...  ]

END
```

The square brackets indicate that declarations and subroutine-definitions are optional in a compound procedure-body.

Data declared in a subroutine is allocated in automatic storage unless the declaration contains a STATIC allocation attribute.

As an example of the compound form of a procedure, suppose you want to dispatch on the value of variable to one of a number of subroutines. You can write a dispatch procedure with that variable as an input parameter as follows:

```
PROC DISPATCH(CODE);
   BEGIN
   ITEM CODE INTEGER S;
   CASE CODE**2;
     (DEFAULT): ;
     (Ø):           ACTION1;
     (1:25):        ACTION2;
     (26:1ØØØ):     ACTION3;
     (1ØØ1:25ØØ):   ACTION4;
     (25Ø1:56ØØ):   ACTION5;
   END
```

The procedure DISPATCH has one formal parameter CODE. The parameter CODE is declared within the procedure to be a signed integer. The statement of the procedure is a case-statement that uses the square of the parameter CODE to select one of five different action routines.

15.1.3.1 Formal Parameters

The formal parameters are given in formal-list. The parameters in formal-list are divided into input parameters and output parameters by a colon, as follows:

    [input-parameter ,...] [: output-parameter, ...]

The square brackets indicate that formal-list can consist of only input-parameters, only output-parameters, or both kinds. If the list contains only output-parameters, the colon must be present to indicate that they are output-parameters. The formal-list, which is enclosed in parentheses, cannot be null; it must contain at least one parameter.

Formal parameters and actual parameters are discussed later in this chapter, after the discussion of procedure-calls.

15.1.4 Procedure-Calls

A procedure-call is a statement. It causes the invocation of the associated procedure and the association of the actual parameters of the call with the formal parameters of the definition.

The form of the procedure-call is:

    procedure-name [(actual-list)] [abort-phrase] ;

The square brackets indicate that both the parenthesized list of actual parameters and the abort-phrase are optional.

The abort-phrase provides a label to which control is sent if an abort-statement is encountered during the execution of the procedure. The abort-phrase has the form:

    ABORT statement-name

A detailed discussion and examples of the abort-phrase are given later in this chapter.

The simplest form of the procedure-call is used for a procedure that is defined without parameters. That is:

```
procedure-name ;
```

For example, to call the procedure TABMULT, which was declared earlier in this chapter without parameters, you can write:

```
TABMULT;
```

The execution of the procedure multiplies each item of PHASE1 by the corresponding item of PHASE2 and saves the product in the corresponding item of PHASES.

If the procedure has parameters, then the procedure-call consists of the procedure-name followed by the parenthesized list of actual parameters, as follows:

```
procedure-name ( actual-list );
```

For example to call the DISPATCH routine, which was declared earlier in this chapter with a single parameter, you can write:

```
DISPATCH(4);
```

The execution of the procedure squares the input parameter and uses that result in a case-statement. This call results in ACTION2 being called by the DISPATCH procedure.

15.1.4.1 Actual Parameters

The parameter list supplied with a subroutine invocation defines the actual parameters to be used for that invocation.

As in the formal parameter list, the actual parameters are divided into input and output parameters by a colon, as follows:

```
[ input-actual, ... ] [: output-actual ,... ]
```

The square brackets indicate that the list of actual parameters can consist of only input parameters, only output parameters, or a combination of both. The actual-list, which in enclosed in parentheses, cannot be null; it must contain at least one paramter.

An input-actual can be a formula, a statement-name, a procedure or function-name, or a block-reference. An output-actual must be a variable.

The relationship between the actual parameters and the formal parameters is discussed later in this chapter.

15.2 FUNCTIONS

A function is different from a procedure in the following ways:

- The function-definition must contain a type description for the result. The function returns a value of that type.
- The function-call is used in a formula (or as a formula) and cannot be used as a statement.
- A function-call cannot contain an abort-phrase.

15.2.1 Function Definitions

A function-definition has the following form:

```
    PROC function-name [ use-attribute ] [ ( formal-list ) ]

                             type-description ;

       procedure-body
```

The type-description defines the type of the return value of the function. The return value must be determined during execution of the function-body by an assignment to the name of the function.

The name of the function is used to designate the return value. The name of the function, when used to designate the return value, can be used only on the left-hand-side of an assignment statement within the function-body. When the name of the function is used on the right-hand-side of an assignment statement or in other permissable contexts for a formula, it designates a recursive function-call.

For example, suppose you want to write a function that gets the factorial of its argument. The factorial is defined for non-negative integers as follows:

```
factorial(Ø) = 1
factorial(n) = 1*...(n-1)*n
```

You can write the following function:

```
PROC FACTORIAL(ARG) U;
   BEGIN
   ITEM ARG U;
   ITEM TEMP U;
   TEMP = 1;
   FOR I:2  BY 1 WHILE I<=ARG;
      TEMP = TEMP*I;
   FACTORIAL = TEMP;
   END
```

The value of the function is an unsigned integer, as indicated by the type-description following the parameter list. The return value is set in the last statement of function-body when TEMP is assigned to FACTORIAL. Observe that TEMP is necessary because the name of the function can be used to designate the return value only on the left-hand-side of an assignment statement within the function-body.

15.2.2 Function-Calls

A function-call can be used as a formula or as an operand in a formula. The form of the function-call is:

```
function-name [ ( actual-list ) ]
```

If the function is defined without parameters, then the function-call is simply the function-name. If the function has parameters, the function-call is the function-name followed by the parenthesized list of actual parameters.

For example, suppose you want to calculate the combination of NN obects taken KK at a time. You can use the factorial function as follows:

```
C2 = FACTORIAL(NN)/(FACTORIAL(NN)-FACTORIAL(KK));
```

The factorial function is called three times in this statement.

15.3 PARAMETERS

The parameters given in a subroutine-definition are called formal parameters because they represent the parameters for the purpose of defining the computations to be performed by the subroutine using the parameters. The parameters given in a subroutine-call are called actual parameters because they are the parameters for that invocation of the subroutine.

15.3.1 Input and Output Parameters

A formal input parameter designates a parameter that receives a value from the corresponding actual parameter. A formal output parameter designates a parameter that receives a value from the corresponding actual parameter when the subroutine is called and returns a value to the corresponding actual when the subroutine is terminated in a normal way.

If, in the course of the execution of a procedure, a formal parameter is used in a context in which its value can be altered, then it must be declared as an output parameter. That is, the value of a formal input parameter cannot be changed within a subroutine.

The number of input actual parameters in the call must be the same as the number of input formal parameters in the definition. Similarly, the number of output actuals must be the same as the number of output formals.

The first (leftmost) actual parameter in a call is associated with the first (leftmost) formal parameter in the definition of the given subroutine; the second actual is associated with the second formal; and so on. However, the order in which the actual parameters are evaluated is not specified. Consider the following procedure declaration:

```
PROC TALLY(:A1);
   BEGIN
   ITEM A1 U;
   A1 = A1 + 1;
   END
```

The preceding definition defines a procedure TALLY with a single parameter. The parameter is an output parameter. It is incremented in the procedure body. Consider a call on TALLY:

```
TALLY(:COUNT);
```

If the value of the item COUNT is 5 before the procedure TALLY is called, then the value of item COUNT is 6 after TALLY is executed.

TALLY is an unrealistically simple function. Usually a function involves more computation. Incrementation like this can be accomplished by a simple assignment statement or, in some cases, by a define-call, as will be seen in Chapter 18.

15.3.2 Parameter Binding

The way in which a formal parameter is bound depends on its type and input/output status. JOVIAL (J73) uses three types of binding: value, value-result, and reference.

A formal input parameter that is an item is bound by value. A formal output parameter that is an item is bound by value-result. A formal parameter that is a table or block is bound by reference.

For all three types of binding, actual parameter values or the location of actual parameter values are evaluated when the subroutine is invoked and are not reevaluated while the subroutine is being executed.

15.3.2.1 Value Binding

If a formal parameter is bound by value, it denotes a distinct object of the type specified in the formal parameter declaration. When the subroutine is called, the value of the actual parameter is assigned to that object.

For example, suppose you have the following procedure-declaration:

```
PROC RUNTIMER(ARG1);
   BEGIN
   ITEM ARG1 U;
   FOR I:Ø BY 1 WHILE I < ARG1**2;
      CORRELATE(ARG1,I);
   END
```

Now suppose you call the procedure:

```
RUNTIMER(CLOCK1);
```

The formal parameter ARG1 is assigned the value of CLOCK1 when the procedure is called.

15.3.2.2 Value-Result Binding

If a formal parameter is bound by value-result, it denotes a distinct object of the type specified in the formal parameter declaration to which the value of the actual is assigned when the subroutine is called. In addition, when the subroutine is exited normally, the value of the formal is assigned to the corresponding actual. If the subroutine is terminated in an abnormal way, the value of the formal is not assigned to the actual. Normal and abnormal returns from subroutines are discussed later in this chapter.

Suppose you define the following procedure:

```
PROC MINMAX(VECTOR:MIN,MAX);
   BEGIN
   TABLE VECTOR(99);
      ITEM V1 U;
   ITEM MIN U;
   ITEM MAX U;
   MIN, MAX = V1(Ø);
   FOR I : 1 BY 1 WHILE I <= 99;
      BEGIN
      IF V1(I) < MIN;
          MIN = V1(I);
      IF V1(I) > MAX;
          MAX = V1(I);
      END
   END
```

Now suppose you call the procedure:

```
MINMAX(RETURNS:RMIN,RMAX);
```

The procedure MINMAX finds the minimum and maximum values in the table RETURNS and, on completion, sets the value of RMIN to the minimum value (MIN) and RMAX to the maximum value (MAX).

15.3.2.3 Reference Binding

If a formal parameter is bound by reference, the actual parameter and the formal parameter denote the same physical object. Any change in the formal parameter entails an immediate change in the value of the actual parameter.

Suppose you want to square the items of a table and then calculate the sum of the pairwise quotients. The item SIZE gives the number of entries in the table currently in use. SIZE is always an even number. You can write:

```
PROC MEAN(:ARGBLOCK);
   BEGIN
   BLOCK ARGBLOCK
      BEGIN
      ITEM SIZE U;
      ITEM SUM U;
      TABLE ARGTAB(1:1000);
        ITEM VALUE S;
      END
   SUM = 0;
   FOR I : 1 BY 2 WHILE I < SIZE;
      BEGIN
      IF VALUE(I+1) = 0;
         ABORT;
      VALUE(I) = VALUE(I)**2;
      VALUE(I+1) = VALUE(I+1)**2;
      SUM = SUM + VALUE(I)/VALUE(I+1);
      END
   END
```

Suppose STATISTICS is a block that is declared as follows:

```
BLOCK STATISTICS;
   BEGIN
   ITEM STATSIZE U = 1Ø;
   ITEM STATSUM U;
   TABLE STATTAB(999);
      ITEM STATVALUE S = 2,4,3,4,8,6,9,Ø,11,3;
   END
```

Suppose you call the procedure MEAN with the actual parameter STATISTICS, as follows:

```
MEAN(:STATISTICS);
```

The block STATISTICS is bound by reference to the formal parameter ARGBLOCK. Each change to SUM results in an immediate change to STATSUM. Similarly, a change in VALUE(I) results in a change in STATVALUE(I). If the procedure terminates abnormally as a result of finding a zero value in the table, STATSUM has the value computed up to that point and the values of the table STATTAB are changed up to the point at which the zero quotient was encountered.

Assuming the item STATSIZE and the table STATTAB have the values assigned in the preset, the procedure terminates abnormally when I is 9. The values of STATSUM and STATTAB on termination are:

Item	Value
STATSUM	4/16 + 9/16 + 64/36
STATVALUE(1)	4
STATVALUE(2)	16
STATVALUE(3)	9
STATVALUE(4)	16
STATVALUE(5)	64
STATVALUE(6)	36
STATVALUE(7)	9
STATVALUE(8)	Ø
STATVALUE(9)	11
STATVALUE(1Ø)	3

15.3.3 Parameter Data Types

The data type of an actual parameter must match the data type of the corresponding formal parameter. The rules for type matching of actual and formal parameters depend on the data types of the parameters:

Items -- The data type of an actual parameter must be compatible with the data type of the corresponding formal parameter.

Tables -- The data type of the actual and formal parameter must be equivalent. The attributes and allocation order of all components of the table must be equivalent.

Blocks -- The data type of a block actual matches the data type of a block formal if (1) the types and order of the components match exactly, (2) an !ORDER directive is either present or absent in both, and (3) overlay declarations in both blocks have the same effect.

Data type equivalence and compatibility are discussed in detail in Chapter 13 on "Conversion".

15.3.4 Parameter Declarations

All parameters must be declared within a subroutine. A formal input parameter can be a data-object, a label, or a subroutine. A formal output parameter can be a data-name only.

A formal parameter cannot be declared to be a constant or a type. Declarations of formal parameters must not contain allocation specifiers or initial values. Formal parameters cannot be declared to be external.

The following sections consider the declarations of data-names, statement-names, and subroutine-names.

15.3.4.1 Data Name Declarations

If a formal parameter is a data name, it is declared by an item, table, or block declaration. The only difference between the form of a data declaration for a data object and that for a formal parameter occurs in the case of a table. A formal parameter that is a table can be declared with asterisk(*) dimensions. The bounds of a table declared in this way are determined from the actual parameter on each invocation of the subroutine.

The use of the asterisk dimensions allows a subroutine to handle tables with different size dimensions. With this capability, general purpose subroutines for table manipulation can be written.

For example, suppose you want to write a procedure that clears to zero the items of a two dimensional table. You can write the following:

```
PROC ZERO(:TABNAME);
   BEGIN
   TABLE TABNAME(*,*);
      ITEM TABENT U;
   FOR I:LBOUND(TABNAME,Ø)  BY 1 WHILE I<=UBOUND(TABNAME,Ø);
     FOR J:LBOUND(TABNAME,1) BY 1 WHILE J<=UBOUND(TABNAME,1);
        TABENT(I,J) = Ø;
   END
```

You can call the procedure ZERO with any two dimensional table. The LBOUND and UBOUND built-in functions provide the lower and upper bounds of the table that is the actual parameter of the call. More information on these built-in functions is given in Chapter 12.

For example, suppose you have the following declarations:

```
TABLE SCORE(1:2Ø,1:3);
   ITEM GRADE U;
TABLE RESULTS(99,4);
   ITEM RES U;
```

You can call ZERO as follows:

```
ZERO(SCORE);
ZERO(RESULTS);
```

The first call on ZERO sets the sixty items of SCORE to zero. The second call sets the five hundred items of RESULTS to zero.

15.3.4.2 Statement Name Declarations

If a formal-parameter is a statement-name, it is declared by a statement-name declaration. The form of a statement-name declaration is:

```
LABEL statement-name ,... ;
```

A GOTO statement to a label that is a formal parameter results in the subroutine being exited and control being sent to the label that is supplied as the actual parameter.

Statement-name parameters are useful for subroutines that have more than one possible error exit. For example, consider the following subroutine:

```
PROC VERIFY(TAB,L1,L2:SUM);
   BEGIN
   TABLE TAB(*);
      ITEM TABENT F;
   LABEL L1,L2;
   ITEM SUM F;
   SUM = Ø.Ø;
   FOR I : LBOUND(TAB,Ø) BY 1 WHILE I <= UBOUND(TAB,Ø);
      BEGIN
      IF TABENT(I) < THRESHOLD;
         GOTO L1;
      SUM = SUM +  TABENT(I)**2
      IF SUM > MAXSUM;
         GOTO L2;
      END
  END
```

Suppose you call the procedure VERIFY as follows:

```
VERIFY(NEWDATA,ERROR1,ERROR5:NEWSUM);
```

The procedure VERIFY is terminated abnormally under two separate conditions. If an entry in NEWDATA is less than THRESHOLD, the procedure is terminated abnormally and control is sent to the label ERROR1. If SUM is greater than MAXSUM, the procedure is terminated abnormally and control is sent to the label ERROR5.

The use of a statement label formal parameter to exit a subroutine constitutes an abnormal termination from the subroutine. Subroutine termination is discussed in detail later in this chapter.

15.3.4.3 Subroutine Declarations

If a formal parameter is a subroutine-name, it is declared by a subroutine-declaration. A subroutine-declaration contains the information necessary to describe a call on the subroutine.

The form of a subroutine-declaration is:

```
PROC procedure-name  [ use-attribute ] [ ( formal-list ) ]

                       type-description ;

     parameter-declaration
```

If the subroutine has parameters, then a declaration must be given for each parameter. If the subroutine does not have any parameters, then a null declaration must be given instead of the parameter declarations. No other declarations can be given in a subroutine-declaration. Declarations of local data, as well as the executable statements and any local subroutine-definitions, are not given in a subroutine-declaration; they can appear only in the subroutine-definition.

If the subroutine-declaration includes a type-description, then it declares a function; otherwise, it declares a procedure.

As an example of the use of a subroutine parameter, suppose that in the VERIFY subroutine just given you want to call a subroutine on an error condition instead of transferring out to a label. You can modify the VERIFY routine as follows:

```
PROC VERIFY(TAB,SUB1,SUB2:SUM);
   BEGIN
   TABLE TAB(*);
      ITEM TABENT F;
   PROC SUB1;
      BEGIN
      END
   PROC SUB2;
      BEGIN
      END
   ITEM SUM F;
   SUM = Ø.Ø;
   FOR I:LBOUND(TAB,Ø) BY 1 WHILE I <= UBOUND(TAB,Ø);
      BEGIN
      IF TABENT(I) < THRESHOLD;
         SUB1;
      SUM = SUM +  TABENT(I)**2;
      IF SUM > MAXSUM;
         SUB2;
      END
```

Suppose you call the procedure VERIFY as follows:

```
VERIFY(NEWDATA,LOWDATA,OVERFLOW:NEWSUM);
```

If an entry within the table NEWDATA is less than THRESHOLD, the procedure LOWDATA is invoked. If SUM is greater than MAXSUM, OVERFLOW is invoked.

15.4 THE USE-ATTRIBUTE

Use-attribute is used to designate a subroutine as either reentrant or recursive.

Use-attribute is one of the following reserved words:

RENT

REC

The use-attribute follows the subroutine-name as indicated earlier in this chapter. If a subroutine is recursive, it must be declared with the REC use-attribute. If it is reentrant, it must be declared with the RENT use-attribute.

15.4.1 Recursive and Reentrant Subroutines

A recursive subroutine is one that calls itself, either directly or indirectly. A reentrant subroutine is one that can be called from several different concurrent tasks.

The compiler must use dynamic storage allocation for a recursive subroutine since the maximum number of recursive invocations and hence the maximum number of copies of automatic data cannot be determined at compile time. If the compiler knows the maximum number of separate tasks that can invoke a reentrant subroutine in a given system, it can allocate storage for the subroutine statically. In general, though, the compiler dynamically allocates storage for reentrant subroutines.

As an example of a recursive function, consider the following, which computes the factorial.

```
PROC RFACT REC (ARG) U;
   BEGIN
   ITEM ARG U;
   IF ARG = Ø;
      RFACT = 1;
   ELSE
      RFACT = RFACT(ARG-1)*ARG;
   END
```

The function RFACT computes the factorial recursively instead of iteratively as did the function FACTORIAL given earlier. The function-declaration contains the use-attribute REC to indicate that it is used recursively.

This function illustrates the use of recursion clearly. In practice, however, a function like this is not written recursively because the computation is too simple to justify the overhead associated with the repetitive function calling mechanism. In the above example, dynamically allocated memory is required for every integer from 1 to the value of ARG, since there is a separate function call for each of these values.

The function RFACT is obviously recursive because it calls itself. Some subroutines are less obviously recursive because they call other subroutines that, in turn, call them.

15.5 SUBROUTINE TERMINATION

The execution of a subroutine is terminated either normally or abnormally.

The execution of a subroutine is terminated in a normal way by one of the following:

- A return-statement
- The execution of the last statement in the subroutine

When a subroutine is terminated in a normal way, the value-result output parameters are set.

The execution of a subroutine is terminated in an abnormal way by one of the following:

- An abort-statement
- A goto-statement to a statement-name supplied as a parameter
- A stop-statement

When a subroutine is terminated abnormally, the value-result output parameters are not set.

15.5.1 Return-Statements

The return-statement is used to effect a normal return from a subroutine. When a return-statement is executed, the execution of the subroutine is terminated, any parameters that have value-result binding are set, and control returns to the point following the subroutine-call.

Suppose you want to search for a particular character string in a table of character strings. You want to stop the search either when you find the character string or when you reach the end of the table. You can use a return-statement for the case in which the character string match is found, as follows:

```
PROC SEARCH(TABNAME,STRING:POSITION);
   BEGIN
   TABLE TABNAME(999);
      ITEM TABSTRING  C 10;
   ITEM STRING C 10;
   ITEM POSITION U;
   FOR POSITION : 0 BY 1 WHILE POSITION  < 1000;
      IF TABSTRING(POSITION) = STRING;
         RETURN;
   NOTFOUND(STRING);
   END
```

If the character string STRING is found in the table TABNAME, the RETURN is executed and the output parameter POSITION gives the entry number in the table where the match occurred. If the character string is not found, the procedure NOTFOUND is called and the output parameter POSITION contains the value 1000.

A return-statement causes a return only from the subroutine in which it is given, not from any subroutines in which the subroutine containing the return is nested.

15.5.2 Abort-Statements

The abort-statement is used to effect an abnormal return from a subroutine. When an abort-statement is executed, control passes to the statement named in the abort-phrase of the most recently executed, currently active procedure call that has an abort-phrase. All intervening subroutine invocations are terminated. No value-result parameters are set.

For example, suppose you want to cause an abnormal termination from the FACTORIAL function if the value of the argument is larger than a specified value. You can include an abort-statement, as follows:

```
PROC FACTORIAL(ARG) U;
   BEGIN
   ITEM ARG U;
   ITEM TEMP U;
   IF ARG > MAXARG;
      ABORT;
   TEMP = 1;
   FOR I : ARG BY -1 WHILE I>0;
      TEMP = TEMP*I;
   FACTORIAL = TEMP;
   END
```

Suppose further that you have another function that gets the combinations of n things taken k at a time using the factorial routine, as follows:

```
PROC COMBINATIONS(NN,KK) U;
   BEGIN
   ITEM NN U;
   ITEM KK U;
   COMBINATIONS = FACTORIAL(NN)/(FACTORIAL(NN)-FACTORIAL(KK));
   END
```

And suppose you want to take the combinations of objects and trials in a table using the following procedure:

```
PROC QUANTIFY(OBJECTS,TRIALS,THRESHOLD:BELOW,EQUAL,ABOVE);
   BEGIN
   TABLE OBJECTS(99);
      ITEM OBJ U;
   TABLE TRIALS(99);
      ITEM TR U;
   ITEM THRESHOLD U;
   ITEM BELOW U;
   ITEM EQUAL U;
   ITEM ABOVE U;
   BELOW, EQUAL, ABOVE = 0;
   FOR I : 0 BY 1 WHILE I <100;
      IF COMBINATIONS(OBJ(I),TR(I)) < THRESHOLD;
         BELOW=BELOW+1;
      ELSE
         IF COMBINATIONS(OBJ(I),TR(I)) = THRESHOLD;
            EQUAL = EQUAL + 1;
         ELSE
            ABOVE = ABOVE + 1;
   END
```

Suppose you call QUANTIFY as follows:

```
QUANTIFY(HITS,GAMES:SUB,EQ,SUPER) ABORT ERROR22;
```

If any of the values in the tables HITS or GAMES results in the factorial being given an argument that exceeds MAXARG, the ABORT statement in the FACTORIAL function is executed and the FACTORIAL function is exited, the COMBINATIONS function is exited, and the QUANTIFY procedure is exited. Since the call of the QUANTIFY procedure has an abort-phrase, control is then sent to the statement labelled ERROR22.

Observe that the name ERROR22 does not need to be known in the scope in which the abort-statement appears. It need be known only in the scope in which the abort-clause on the procedure-call appears.

15.5.3 Goto-Statements

A goto-statement to a formal label parameter transfers control from a subroutine prematurely. If a GOTO to a formal parameter is executed, control is sent to the statement whose label was passed as an actual parameter in the subroutine call.

Value-result parameters are not set if a subroutine is exited by transferring to a label parameter.

15.5.4 Stop-statements

A stop-statement stops the execution of the entire program at the point it is given. Value-result parameters are not set if a stop-statement is executed in a subroutine.

15.6 MACHINE SPECIFIC SUBROUTINES

Machine specific subroutines are those subroutines provided by an implementation to enable programs to invoke single machine instructions peculiar to the given machine. In general, subroutines are provided for machine instructions that cannot be executed through the language.

15.7 THE INLINE-DECLARATION

The inline-declaration directs the compiler to insert the object code for the body of the subroutines named at the points of their invocation instead of generating object code to call them. The form is:

```
INLINE subroutine-name ,... ;
```

The effect of an inline-declaration extends for just the name scope in which the inline-declaration apears. It does not affect calls in enclosing scopes. Subroutine-names whose definitions appear in other modules cannot be used in an inline-declaration.

Suppose you have the following statements:

```
ITEM COUNT U;
TABLE DX1(999);
   ITEM DXCOUNT U;
INLINE TALLY;
COUNT = Ø;
FOR I:Ø BY 1 WHILE I<1ØØØ;
   IF DXCOUNT(I) < THRESHOLD;
      TALLY(:COUNT);
PROC TALLY(:ARG);
   BEGIN
   ITEM ARG U;
   ARG = ARG +1;
   END
```

The code generated for the loop-statement is the same as the code generated for the following loop-statement:

```
FOR I:Ø BY 1 WHILE I<1ØØØ;
   IF DXCOUNT(I) < THRESHOLD;
      COUNT = COUNT + 1;
```

If any actual parameters of a subroutine that is declared inline are constants, the inline expansion may cause some formulas to be evaluable at compile-time. In such a case, compile-time evaluation of these formulas will occur and any corresponding error messages will be generated.

Inline subroutines can contain subroutine-calls, which can in turn be inline. However, an inline subroutine cannot contain subroutine-definitions. Also, it is illegal to have an inline subroutine invocation of a subroutine that is already being expanded inline (that is, recursive invocation of inline subroutines is not allowed).

Chapter 16

EXTERNALS AND MODULES

A JOVIAL (J73) program consists a main program module and one or more compool or procedure modules. Execution of the program begins at the first statement of the main-program module and continues until the last statement of the program or a stop-statement is executed.

A module is the smallest entity in the language that can be compiled separately. The modules of a program are compiled separately and subsequently bound together for execution as a unit.

Communication between separately compiled modules is accomplished by external names. An external name is a name declared in one module and used in one or more other modules.

This chapter begins by discussing external declarations. Then it considers the different types of module. Finally, the subject of scope is revisited in the context of module compilation and module communication is considered.

16.1 EXTERNAL DECLARATIONS

An external declaration declares an external name, that is: a name with the external attribute. An external name can be made available for use in other modules.

Two kinds of external declarations are defined in JOVIAL (J73), DEF-specifications and REF-specifications. A DEF-specification declares an external name and allocates storage for that name. A REF-specification provides information about an external name that is declared in another module by a DEF-specification.

Each DEF-specification identifies a unique object that can be referenced by a REF-specification in any number of modules. It is illegal to have a second DEF-specification with the same name anyplace in the entire program, unlike other data objects, not declared with a DEF-specification, which can have identifical names providing scope rules prevent conflict.

The following sections describe the DEF- and REF-specifications in detail.

16.1.1 DEF-Specifications

A DEF-specification is used to declare a name that can be used in other modules. A DEF-specification can be either simple or compound.

16.1.1.1 Simple DEF-Specifications

The form of a simple DEF-specification is:

```
DEF declaration
```

Consider the following simple DEF-specification:

```
DEF TABLE GRID (20,20);
        BEGIN
        XCOORD U 5;
        YCOORD U 5;
        END
```

This declaration declares the table GRID and associates with it the external attribute.

16.1.1.2 Compound DEF-Specifications

The form of a compound DEF-specification is:

```
     DEF BEGIN

         declaration
         ...

         END
```

Consider the following compound DEF-specification:

```
     DEF
        BEGIN
        ITEM RATE U 10;
        ITEM TIME U 15;
        TABLE STOCKS(100);
           BEGIN
           ITEM NAME C 6;
           ITEM QUOTE C 3;
           END
        END
```

This declaration declares the items RATE and TIME and the table STOCKS and associates the external attribute with each of those data objects.

A DEF-specification can be used to declare an item, table, block, or statement name. A DEF-specification can be used to define a subroutine in a main program or procedure module, but not in a compool module.

A DEF-specification for a statement name makes the address of the statement available for linkage purposes. The statement name, however, cannot be used as the target of a GOTO statement that is in another module, or in any other way to cause control to transfer outside the given scope.

16.1.1.3 Allocation

Data or subroutines declared by a DEF-specification in a module are physically allocated in that module.

A DEF-specification can only be used with data objects that are allocated statically. Data declared external in a subroutine, therefore, must have a STATIC allocation-spec.

For example, to declare the external item FLAGS within the procedure MONITOR, you can write:

```
PROC MONITOR(STATE);
   BEGIN
   DEF ITEM FLAGS STATIC B 5;
   ...

   END
```

The item FLAGS is declared as an external name. The declaration includes the STATIC allocation-spec because the declaration is given within the subroutine MONITOR.

16.1.2 REF-Specifications

A REF-specification is used to reference a name that is declared by a DEF-specification in another module. A REF-specification can be either simple or compound. It has one of the following two forms:

```
REF declaration

REF BEGIN

    declaration
    ...

    END
```

A REF-specification can be used to declare an item, table, block, or subroutine.

A REF-specification is used to make available within a compool-module information about externals declared by a DEF-specification in other modules.

For example, you can use a REF-specification in the compool-module SPECS to make the subroutine AVERAGE available through that module. AVERAGE is defined in a procedure-module as follows:

```
START
DEF PROC AVERAGE (TABSIZE,TABNAME: RESULT);
   BEGIN
   ITEM TABSIZE U;
   ITEM RESULT U;
   TABLE TABNAME(1:100);
      ITEM ENTRY U;
   RESULT = 0;
   FOR I : 1 BY 1 WHILE I < TABSIZE;
      RESULT = RESULT + ENTRY(I);
   RESULT = RESULT/TABSIZE;
   END
TERM
```

You can include a REF-specification in the compool-module SPECS as follows:

```
START COMPOOL SPECS;
   DEF ITEM RATE U 10;
   DEF ITEM FLAGS B 3;
   DEF TABLE SUBSCRIBERS(100);
          BEGIN
          ITEM NAME C 5;
          ITEM ADDRESS C 20;
          ITEM CITY C 10;
          ITEM STATE C 3;
          END
   REF PROC AVERAGE(TABSIZE,TABNAME: RESULT);
          BEGIN
          ITEM TABSIZE U;
          ITEM RESULT U;
          TABLE TABNAME(1:100);
             ITEM ENTRY U;
          END
   TERM
```

The REF-specification in the compool-module contains the procedure-declaration for AVERAGE. The DEF-specification in the procedure-module contains the procedure-definition. Since external subroutines cannot be defined in a compool-module, a REF-specification is the only way information about the subroutine can be given in the compool.

Data can also be declared physically in a non-compool module and made available in the compool by a REF-specification.

A name declared in a REF-specification must agree in name, type, and all other attributes with the name declared in the DEF-specification. The compiler checks the agreement of REF- and DEF-specifications under certain circumstances, which are discussed later in this chapter in connection with the compool-directive.

Presets must not be given in a REF-specification for an item or a table. Presets can be given in a REF-specification for a block only in one special case, which is discussed in Chapter 19 on "Advanced Topics".

16.1.3 Constant Data

A constant declaration cannot be declared external directly with either a DEF- or REF-specification. However, a block containing a constant declaration can be made external.

The following is a valid declaration:

```
DEF BLOCK PSEUDOBLOCK;
       BEGIN
       CONSTANT ITEM PI F = 3.14159;
       END
```

16.2 MODULES

A module can be any of the following:

Main-Program-Module

Compool-Module

Procedure-Module

When a module is compiled, it exists within the two additional scopes described earlier in Chapter 4, the system scope and the compool scope.

The following sections consider the form and content of each of the three kinds of module. Then module compilation is considered.

16.2.1 Main Program Module

A main-program-module contains a program-body and an optional sequence of non-nested-subroutines. The form of the main-program-module is:

```
START PROGRAM program-name ;

    program-body

    [ non-nested-subroutine ... ]

TERM
```

A program-body is the same as a subroutine-body. It is either a single statement or a sequence of declarations, statements, and subroutine-definitions, as follows:

```
              BEGIN

              [ declaration   ...  ]

              statement  ...

              [ subroutine-definition  ... ]

  [ label ... ]

              END
```

The declarations and subroutine-definitions are optional, bu' '' program-body must contain at least one executable statement.

A non-nested-subroutine is a subroutine definition that can be made external by the addition of the DEF reserved word, as follows:

```
[ DEF ] subroutine-definition
```

A non-nested-subroutine can contain nested subroutines.

When a program is executed, the statement or statements of the program-body of the main-program-module are executed, starting with the first and continuing until the execution is complete.

Consider the following main-program-module:

```
START PROGRAM SEARCH;
   BEGIN
   TYPE KEY STATUS (V(RED),V(GREEN),V(YELLOW));
   TYPE DBASE
      TABLE (1000);
         BEGIN
         ITEM CODE KEY;
         ITEM VALUE U;
         END
   TABLE DATA DBASE;
   ITEM CURVAL U;
   GETVALUE(DATA);
   CURVAL=RETRIEVE(V(RED));
   PROC RETRIEVE(ARG1) U;
      BEGIN
      ITEM ARG1 KEY;
      FOR I:0 BY 1 WHILE I<=1000;
        IF CODE(I) = ARG1;
           RETRIEVE = VALUE(I);
      ERROR(20);
      END
   END
DEF PROC GETVALUE(ARGTAB);
   BEGIN
   TABLE ARGTAB DBASE;
   ...
   END
DEF PROC ERROR(ERRNO);
   BEGIN
   ITEM ERRNO U;
   ...
   END
TERM
```

This main-program-module consists of a program-body and two non-nested subroutines. The program-body contains two type-declarations, a table-declaration, an item-declaration, two statements, and a nested subroutine-definition.

This main-program-module is independent and could be compiled and executed. The following sections will illustrate how some of the information in this main-program-module can be put in other modules.

16.2.2 Compool-Modules

A compool-module provides for the communication of names between separately compiled modules. A compool-module can contain only declarations. The form is:

```
START COMPOOL compool-name ;

      declaration ...

TERM
```

The following kinds of declaration are allowed in a compool-module:

```
constant declaration
type declaration
define declaration
overlay declaration
DEF-specification  for a data or statement name declaration
REF-specification  for a data or subroutine declaration
```

Constant declarations were discussed in Chapters 6 and 7. Type declarations were discussed in Chapter 9. Define delarations will be discussed in Chapter 18 and overlay declarations in Chapter 19. DEF- and REF-specifications are described later in this chapter.

As an example of a compool-declaration, consider the following:

```
START COMPOOL TYPEDEFS;
   TYPE KEY STATUS (V(RED),V(GREEN),V(YELLOW));
   TYPE DBASE
      TABLE (1000);
         BEGIN
         ITEM CODE KEY;
         ITEM VALUE U;
         END
   TERM
```

The compool TYPEDEFS contains two type declarations, one for the item type KEY and one for the table type DBASE.

The information in a compool-module is made available to the module being compiled by a !COMPOOL directive. !COMPOOL directives are given immediately following the START in the module being compiled. A !COMPOOL directive makes available either the declarations for a set of given names or all the declarations in the compool, depending on its form. The !COMPOOL directive is discussed in more detail a little later in this chapter and fully in Chapter 17, "Directives".

With this compool, the main-program-module given earlier could be written using a !COMPOOL directive to supply the necessary type-declartions, as follows:

```
START !COMPOOL ('TYPEDEFS');
   PROGRAM SEARCH;
      BEGIN
      TABLE DATA DBASE;
      ITEM CURVAL U;
      GETVALUE(DATA);
      CURVAL=RETRIEVE(V(RED));
      PROC RETRIEVE(ARG1) U;
         BEGIN
         ITEM ARG1 KEY;
         FOR I:Ø BY 1 WHILE I<=1ØØØ;
            IF CODE(I) = ARG1;
                RETRIEVE = VALUE(I);
         ERROR(2Ø);
         END
      END
   DEF PROC GETVALUE(ARGTAB);
      BEGIN
      TABLE ARGTAB DBASE;
       ...
      END
   DEF PROC ERROR(ERRNO);
      BEGIN
      ITEM ERRNO U;
      ...
      END
TERM
```

The fact that the compool name TYPEDEFS is parenthesized indicates that all the declarations in that compool are made available to the main-program-module. Thus, the declarations for the type KEY and the type DBASE are made available from the compool and can be used, without declaration, in the main-program-module.

These declarations are in the compool scope, as discussed earlier in Chapter 4 on "Declarations and Scope". The scopes of the main-program-module SEARCH can be diagrammed as follows:

```
+system-scope-----------------------------------------------+
|                                                            |
|   +compool-scope------------------------------------+     |
|   |            DBASE, KEY, SEARCH                    |     |
|   |     +module-scope-----------------------------+  |     |
|   |     |                                          |  |     |
|   |     |  +module-body-scope----------------+    |  |     |
|   |     |  |  DATA, CURVAL                    |    |  |     |
|   |     |  |  RETRIEVE                        |    |  |     |
|   |     |  |     +subr-scope---------+        |    |  |     |
|   |     |  |     | ARGTAB            |        |    |  |     |
|   |     |  |     +-------------------+        |    |  |     |
|   |     |  |                                  |    |  |     |
|   |     |  +----------------------------------+    |  |     |
|   |     |  GETVALUE                                |  |     |
|   |     |     +subr-scope--------------+           |  |     |
|   |     |     |  ARG1                  |           |  |     |
|   |     |     +------------------------+           |  |     |
|   |     |  ERROR                                   |  |     |
|   |     |     +subr-scope--------------+           |  |     |
|   |     |     |  ERRNO                 |           |  |     |
|   |     |     +------------------------+           |  |     |
|   |     +------------------------------------------+  |     |
|   |                                                   |     |
|   +---------------------------------------------------+     |
|                                                             |
+-------------------------------------------------------------+
```

The module name SEARCH and the type-names DBASE and KEY are in the compool scope. The names of the non-nested subroutines GETVALUE and ERROR are in the module scope. The names of the data objects DATA and CURVAL and the name of the subroutine RETRIEVE are in the module-body scope. The name of the formal parameter ARGTAB is in the subr-body scope of RETRIEVE, and so on.

Declarations are placed in a compool principally so that they are available for use by more than one module. The use of compools provides a logical program structure that can be readily generalized. In some cases, declarations that are not shared are also placed in a compool for structural purposes.

Suppose, in our example, that the table DATA is needed in several modules. If each module is to use the same table, that table must be located in a compool. Consider the following compool-declaration:

```
     START !COMPOOL ('TYPEDEFS');
        COMPOOL DATABASE;
           BEGIN
           DEF TABLE DATA DBASE;
           DEF ITEM CURVAL U;
           END
      TERM
```

This compool module contains a !COMPOOL directive, which makes the declarations of the compool TYPEDEFS available. Thus, the type-names DBASE and CURVAL do not need to be declared in this module.
The scopes during the compilation of the module DATABASE are as follows:

```
+system-scope-----------------------------------------------+
|                                                           |
|   +compool-scope----------------------------------------+ |
|   |           DBASE, KEY, DATABASE                      | |
|   |    +module-scope-----------------------------+      | |
|   |    |                                         |      | |
|   |    |   +module-body-scope-----------------+  |      | |
|   |    |   | DATA, CURVAL                     |  |      | |
|   |    |   |                                  |  |      | |
|   |    |   +----------------------------------+  |      | |
|   |    |                                         |      | |
|   |    +-----------------------------------------+      | |
|   |                                                     | |
|   +-----------------------------------------------------+ |
|                                                           |
+-----------------------------------------------------------+
```

The main-program module can be written using the compool DATABASE, as follows:

```
START !COMPOOL ('DATABASE');
   PROGRAM SEARCH;
      BEGIN
      GETVALUE(DATA);
      CURVAL=RETRIEVE(V(RED));
      PROC RETRIEVE(ARG1) U;
         BEGIN
         ITEM ARG1 KEY;
         FOR I:Ø BY 1 WHILE I<=1ØØØ;
            IF CODE(I) = ARG1;
               RETRIEVE = VALUE(I);
         ERROR(2Ø);
         END
      END
   DEF PROC GETVALUE(ARGTAB);
      BEGIN
      TABLE ARGTAB DBASE;
       ...
      END
   DEF PROC ERROR(ERRNO);
      BEGIN
      ITEM ERRNO U;
      ...
      END
TERM
```

The data objects DATA and CURVAL are declared in the compool DATABASE and therefore do not need to be declared in the main-program-module. The type-names are needed only for the declaration of the data objects and so the main-program-module does not need to have a !COMPOOL directive for the compool TYPEDEFS. The type-names DBASE and KEY are not known in the main-program-module.

The scopes during the compilation of the main-program-module can now be diagrammed as follows:

```
+system-scope----------------------------------------------+
|                                                          |
|   +compool-scope--------------------------------------+  |
|   |           DATA, CURVAL, SEARCH                    |  |
|   |    +module-scope------------------------------+   |  |
|   |    |                                          |   |  |
|   |    |  +module-body-scope------------------+   |   |  |
|   |    |  |                                   |   |   |  |
|   |    |  |  RETRIEVE                         |   |   |  |
|   |    |  |     +subr-scope---------+         |   |   |  |
|   |    |  |     | ARGTAB            |         |   |   |  |
|   |    |  |     +-------------------+         |   |   |  |
|   |    |  |                                   |   |   |  |
|   |    |  +-----------------------------------+   |   |  |
|   |    |  GETVALUE                                |   |  |
|   |    |     +subr-scope--------------+           |   |  |
|   |    |     | ARG1                   |           |   |  |
|   |    |     +------------------------+           |   |  |
|   |    |  ERROR                                   |   |  |
|   |    |     +subr-scope--------------+           |   |  |
|   |    |     | ERRNO                  |           |   |  |
|   |    |     +------------------------+           |   |  |
|   |    +------------------------------------------+   |  |
|   |                                                   |  |
|   +---------------------------------------------------+  |
|                                                          |
+----------------------------------------------------------+
```

The next step in simplifying the main-program is to place the subroutines in a module. Subroutine-definitions must be given in a procedure-module.

16.2.3 Procedure-Modules

A procedure-module provides a way in which the subroutines of a program can be compiled separately. A procedure module contains declarations and subroutine definitions. The form is:

```
START

  [ declaration ... ]

  [ [ DEF ] subroutine-declaration ... ]

TERM
```

Any type of declaration can be given in a procedure module.

As an example of a procedure-module, consider the following:

```
START !COMPOOL ('TYPEDEFS');
   DEF PROC GETVALUE(ARGTAB);
      BEGIN
      TABLE ARGTAB DBASE;
       ...
      END
   DEF PROC ERROR(ERRNO);
      BEGIN
      ITEM ERRNO U;
      ...
      END
TERM
```

The procedure module contains two external subroutine definitions. The type-name DBASE is provided by the declaration of DBASE in the compool TYPEDEFS.

In order to make these subroutine definitions available to a another module, a link must be made with a compool, by including a REF-specification in the compool and a !COMPOOL directive in the procedure module.

Suppose we include the REF-specifications in the DATABASE module as follows:

```
START !COMPOOL ('TYPEDEFS');
   COMPOOL DATABASE;
      BEGIN
      DEF TABLE DATA DBASE;
      DEF ITEM CURVAL U;
      REF PROC GETVALUE(ARGTAB);
         TABLE ARGTAB DBASE;
      REF PROC ERROR(ERRNO);
         ITEM ERRNO U;
      END
 TERM
```

Now, we compile the compool module DATABASE in the compool scope which includes the declarations from TYPEDEFS.

We then include a compool-directive for the compool DATABASE in the procedure module, so that the compiler can check the agreement of the corresponding DEF- and REF-specifications.

```
START !COMPOOL 'DATABASE' GETVALUE,ERROR;
      !COMPOOL ('TYPEDEFS');
   DEF PROC GETVALUE(ARGTAB);
      BEGIN
      TABLE ARGTAB DBASE;
       ...
      END
   DEF PROC ERROR(ERRNO);
      BEGIN
      ITEM ERRNO U;
      ...
      END
TERM
```

Then we compile the procedure-module in the compool scope which includes the REF-specifications from the compool DATABASE and the type-declarations from the compool TYPEDEFS.

The scopes for the compilation of the procedure module can be diagrammed as follows:

```
+system-scope-----------------------------------------------+
|                                                           |
|   +compool-scope-------------------------------------+    |
|   |            DBASE, KEY, GETVALUE, ERROR           |    |
|   |    +module-scope-----------------------------+   |    |
|   |    |                                         |   |    |
|   |    |  +module-body-scope-----------------+   |   |    |
|   |    |  |                                  |   |   |    |
|   |    |  | GETVALUE                         |   |   |    |
|   |    |  |   +subr-scope--------------+     |   |   |    |
|   |    |  |   |  ARG1                  |     |   |   |    |
|   |    |  |   +------------------------+     |   |   |    |
|   |    |  | ERROR                            |   |   |    |
|   |    |  |   +subr-scope--------------+     |   |   |    |
|   |    |  |   |  ERRNO                 |     |   |   |    |
|   |    |  |   +------------------------+     |   |   |    |
|   |    |  |                                  |   |   |    |
|   |    |  +----------------------------------+   |   |    |
|   |    +-----------------------------------------+   |    |
|   |                                                  |    |
|   +--------------------------------------------------+    |
|                                                           |
+-----------------------------------------------------------+
```

The compiler checks that the REF-specifications in the compool agree with the DEF-specifications in the module being compiled.

The main-program module now can be written using the compool DATABASE, as follows:

```
START !COMPOOL (DATABASE);
   PROGRAM SEARCH;
      BEGIN
      GETVALUE(DATA);
      CURVAL=RETRIEVE(V(RED));
      PROC RETRIEVE(ARG1) U;
         BEGIN
         ITEM ARG1 KEY;
         FOR I:Ø BY 1 WHILE I<=1ØØØ;
            IF CODE(I) = ARG1;
                RETRIEVE = VALUE(I);
         ERROR(2Ø);
         END
      END
TERM
```

16.3 MODULE COMMUNICATION

As has been illustrated in the preceding sections, modules can communicate by compool directives. If a declaration is to be used in more than one module, it is placed in a compool. Then it can be referenced in each module that needs it by a compool-directive.

A compool-directive can make all the declarations in a given compool available or it can make a selected set of declarations available. A compool-directive that makes all declarations available has the form:

```
! COMPOOL ( compool-file ) ;
```

A compool-directive that makes selected declarations available has the form:

```
! COMPOOL compool-file name ,... ;
```

Only the declarations of those names given are made available in this form.

Compool-file is a character literal designated by the implementation to correspond to a given compool. Name is a name declared in the compool.

Other forms of the compool-directive, as well as a complete discussion of this directive, are given in Chapter 16.

16.3.1 Direct Communication

If communication between modules is accomplished through compool-directives, the compiler provides the declaration of the shared object. If the module using that object does not use it in a manner that is consistent with its declaration, the compiler detects and reports the error.

A REF-specification can be used in one module to directly communicate with another module, but in this case, no checking can be performed. The compiler must assume that the type class and attributes given in the REF-specification are accurate. At link time, the references to the name are bound together, but no check of type or attributes can be made because that information is not available at link time.

Thus, if the REF-specification declares an object of one type and the DEF-specification declares an object of another type, the program that is formed by linking the separately compiled modules is invalid and the results of its execution are unpredictable.

As an example of direct communication, suppose we have a procedure module that contains some external subroutine definitions as follows:

```
START !COMPOOL (TYPEDEFS);
   DEF PROC GETVALUE(ARGTAB);
      BEGIN
      TABLE ARGTAB DBASE;
       ...
      END
   DEF PROC ERROR(ERRNO);
      BEGIN
      ITEM ERRNO U;
       ...
      END
TERM
```

Now, suppose that the module DATABASE does not contain a REF-specification for the subroutine ERROR, but instead the main-program module includes a REF-specification for ERROR, as follows:

```
START !COMPOOL (DATABASE);
   PROGRAM SEARCH;
      BEGIN
      REF PROC ERROR(ERRNO);
         ITEM ERRNO U;
      GETVALUE(DATA);
      CURVAL=RETRIEVE(V(RED));
      PROC RETRIEVE(ARG1) U;
         BEGIN
         ITEM ARG1 KEY;
         FOR I:Ø BY 1 WHILE I<=1ØØØ;
            IF CODE(I) = ARG1;
                RETRIEVE = VALUE(I);
         ERROR(2Ø);
         END
      END
TERM
```

In this case, the REF-specification for ERROR agrees with the DEF-specification, and the resulting program operates correctly. However, suppose the REF-specification indicated that the subroutine ERROR has two arguments. The compiler cannot detect any error, the linker makes the connection and the resulting program is invalid but no indication of its invalidity can be made.

Chapter 17

DIRECTIVES

Directives are used to provide supplemental information to the compiler about the program. Directives affect output format, program optimization, data and subroutine linkage, debugging information, and other aspects of program processing.

Most directives change the way a program is processed without changing the computation performed by the program. Perhaps the simplest example of such a directive is "!EJECT", which starts a new page in the compiler's listing of the program.

In general, directives can appear after the reserved word START and before any statement, declaration, or optionally labelled END. Some directives can only be placed in certain positions.

This chapter describes the directives in detail and indicates the placement of each directive. Each section considers a particular class of directive. The classes of directives and the names and form of the directives in each class are given on the next page.

Class	Directive Form
compool	!COMPOOL (compool-file) ; !COMPOOL compool-file name ,... ;
text	!COPY file ; !SKIP letter ; !BEGIN letter ; !END ;
listing	!LIST ; !NOLIST ; !EJECT ;
initialization	!INITIALIZE ;
allocation-order	!ORDER ;
evaluation-order	!LEFTRIGHT ; !REARRANGE ;
interference	!INTERFERENCE data-name : data-name ,... ;
reducible	!REDUCIBLE ;
register	!BASE data-name register-number ; !ISBASE data-name register-number ; !DROP register-number ;
linkage	!LINKAGE symbol ;
trace	!TRACE (control) name ,... ; !TRACE name ,... ;

17.1 COMPOOL-DIRECTIVES

A compool-directive is used to identify the compool and the set of names from that compool that are to be used in the compool scope for the module being compiled.

Depending on its form, a compool-directive can make available all the declarations in a compool or a selected set of declarations. A compool-directive that provides access to all the definitions declared in the compool gives the compool-file enclosed in parentheses, as follows:

```
!COMPOOL ( compool-file ) ;
```

This form makes available all the declarations in the compool. Declarations used in the named compool that were obtained from other compools by a compool-directive, however, are not made available. This case was illustrated in Chapter 16.

Compool-file is an implementation-designated character literal that identifies the given compool. If compool-file is not given, the compiler assumes an unnamed compool.

A compool-directive that provides access to a selected set of definitions from a compool has the following form:

```
!COMPOOL  compool-file  name ,... ;
```

The module that contains this compool-directive has access only to the declarations of the names given in this directive, plus any additional names that are associated with these names and are automatically included.

17.1.1 Names

The names given in a compool-directive must be declared in the designated compool. Further, a given name cannot be the name of an entity declared in a type-declaration or the name of a formal parameter.

17.1.2 Additional Declarations

Additional declarations are, in some cases, made available to permit the full use of a given name. Additional declarations are provided in the following cases:

- If the name given in a compool-directive is the name of an item, table, or block declared using a type-name, then the declaration of the type-name is also made available, provided it is declared within the compool and not brought in by a compool-directive.

  For a pointer item, the definition of the type-name that is the pointed-to type is also made available, provided it is declared within the compool and not brought in by a compool-directive.

- If the given name is the name of an item within a table, then the table name is also made available.

- If the given name is a table name, the definitions of any status-lists or status-type-names associated with the table's dimensions are also made available, provided they are declared in the compool.

- If the given name is a table type-name or block type-name, the definitions of the components are made available.

- If the given name is a status item name, its associated status-list and status type-name (if any) are also made available, provided they are declared in the compool.

- If the given name is the name of a subroutine, any type-names associated with the subroutine's formal parameters or return value are also made available, provided they are declared in the compool.

17.1.3 Placement

A compool-directive can be given only immediately following the START reserved word of a module or following another compool-directive.

17.1.4 Examples

Suppose you have the following compool:

```
START COMPOOL BSQDATA;
   DEF ITEM HEIGHT U;
   DEF ITEM WIDTH U;
   DEF ITEM LENGTH U;
   DEF TABLE GRID(20,20);
      BEGIN
      ITEM XCOORD U;
      ITEM YCOORD U;
      END
   TERM
```

The following list gives different forms of the compool-directive and indicates the declarations that are made available for each form.

Directive	Available Declarations
!COMPOOL 'BSQDATA' LENGTH;	LENGTH
!COMPOOL 'BSQDATA' LENGTH,WIDTH;	LENGTH, WIDTH
!COMPOOL 'BSQDATA' GRID;	GRID
!COMPOOL 'BSQDATA' (GRID);	GRID, XCOORD, YCOORD
!COMPOOL ('BSQDATA');	LENGTH, HEIGHT, WIDTH, GRID, XCOORD, YCOORD

17.2 TEXT-DIRECTIVES

The text-directives are used to modify the source program. The !COPY text-directive is used to copy the contents of a file into a program at a particular point and the conditional directives are used to permit conditional compilation, by indicating those portions of the program that are and are not to be compiled.

17.2.1 Copy-Directive

The copy-directive names the file that is to be copied into the program at the point where the copy-directive is given. The form of the copy-directive is:

```
!COPY file ;
```

File is a character literal that is an implementation dependent file-name.

17.2.1.1 Placement

The copy-directive can be placed anywhere a directive can be given.

17.2.1.2 Example

An example of the copy-directive is:

```
!COPY 'IDENT.NEW';
```

The compiler replaces the copy-directive by the file named in the directive. A define-call, as will be seen in Chapter 18 lao produces a text replacement. A copy-directive is different from a define-call in that it refers to an external file, it cannot be parameterized, and it can be given only in places where directives can appear.

17.2.2 Conditional-Compilation-Directives

Three conditional-compilation-directives are defined. The forms of the directives are as follows:

```
!SKIP letter ;

!BEGIN  letter ;

!END ;
```

The !SKIP directive identifies the blocks of source program that are to be skipped. The other two directives, the !BEGIN and !END directives delimit the block that is to be included or skipped depending on the !SKIP directives that are included in the program.

17.2.2.1 Placement

The conditional-compilation-directives can be given anywhere a directive can be given. The !SKIP directive must be given before the associated !BEGIN and !END directives. For each !BEGIN directive, a matching !END directive must be given.

17.2.2.2 Examples

Suppose a program includes a computation that can be written either to execute efficiently or to conserve storage. You can include both versions of the computation in your program and choose between the two versions by changing the letter on the !SKIP directive, as follows:

```
START PROGRAM MAIN;
   BEGIN

    ( declarations and statements )

   !SKIP A;
   !BEGIN A;

     ( time efficient computation )

   !END;
   !BEGIN B;

    ( space-efficient computation )

   !END;
   END
TERM
```

If the letter A is used with the !SKIP directive, as shown above, this directive instructs the compiler to omit the conditional block labelled A. As a result, the program uses the space-efficient computation. If the letter B is used with the !SKIP directive, the compiler omits the conditional block labelled B and uses the time-efficient computation.

As another example, suppose you want to select one of two possible functions that produce a random number, based on the !SKIP directive. You can write:

```
START PROGRAM MAIN
   BEGIN
   !SKIP Y;
   ITEM RESULT U;
   ITEM COUNT U;
   !BEGIN X;
   REF PROC RND U;
     BEGIN
     END
   RESULT = RND;
   !END;
   !BEGIN Y
   REF PROC RANDOM U;
     BEGIN
     END
   RESULT = RANDOM;
   !END
   COUNT = Ø;
   CASE RESULT;
      BEGIN
      ( DEFAULT ): ;
      (1:1ØØ):   COUNT = COUNT + 1;
      (1Ø1:5ØØ): COUNT = COUNT + 2;
      (5Ø1:9ØØ): COUNT = COUNT + 3;
      END
   END
TERM
```

The !SKIP directive indicates that the information in the block associated with Y is to be skipped. The program that is compiled then is:

```
START PROGRAM MAIN
   BEGIN
   ITEM RESULT U;
   ITEM COUNT U;
   REF PROC RND U;
      BEGIN
      END
   RESULT = RND;
   COUNT = Ø;
   CASE RESULT;
      BEGIN
      ( DEFAULT ) : ;
      (1:1ØØ):   COUNT = COUNT + 1;
      (1Ø1:5ØØ): COUNT = COUNT + 2;
      (5Ø1:9ØØ): COUNT = COUNT + 3;
      END
   END
TERM
```

Conditional compilation blocks can be nested. If a !SKIP directive indicates that the outer block is to be skipped, then the inner block is processed only to associate BEGIN END pairs. If the outer block is not skipped, then a !SKIP directive can be included to skip an inner block.

Suppose you want to square the result in some cases when you use the function RANDOM. You can associate the squaring of the result with a conditional block as follows:

```
START PROGRAM MAIN;
   BEGIN
   !SKIP X;
   ITEM RESULT U;
   ITEM COUNT U;
   !BEGIN X;
   REF PROC RND U;
      BEGIN
      END
   RESULT = RND;
   !END;
   !BEGIN Y
   REF PROC RANDOM U;
      BEGIN
      END
  RESULT = RANDOM;
  !BEGIN A;
  RESULT = RESULT**2;
  !END;
  !END;
  COUNT = Ø;
  CASE RESULT;
     BEGIN
     ( DEFAULT ): ;
     (1:1ØØ):    COUNT = COUNT + 1;
     (1Ø1:5ØØ): COUNT = COUNT + 2;
     (5Ø1:9ØØ): COUNT = COUNT + 3;
     END
  END
TERM
```

The !SKIP directive instructs the compiler to omit the conditional block associated with X. The program that is compiled is:

```
START PROGRAM MAIN;
   BEGIN
   ITEM RESULT U;
   ITEM COUNT U;
   REF PROC RANDOM U;
      BEGIN
      END
   RESULT = RANDOM;
   RESULT = RESULT**2;
   COUNT = Ø;
   CASE RESULT;
      BEGIN
      ( DEFAULT ) : ;
      (1:1ØØ):    COUNT = COUNT + 1;
      (1Ø1:5ØØ):  COUNT = COUNT + 2;
      (5Ø1:9ØØ):  COUNT = COUNT + 3;
      END
   END
TERM
```

You can then omit the squaring of RESULT by including a !SKIP directive for A, as follows:

```
START PROGRAM MAIN;
   BEGIN
   !SKIP X;
   !SKIP A;
   ITEM RESULT U;
   ITEM COUNT U;
   !BEGIN X;
   REF PROC RND U;
      BEGIN
      END
   RESULT = RND;
   !END;
   !BEGIN Y
   REF PROC RANDOM U;
      BEGIN
      END
   RESULT = RANDOM;
   !BEGIN A;
   RESULT = RESULT**2;
   !END;
   !END
   COUNT = Ø;
   CASE RESULT;
      BEGIN
      ( DEFAULT ): ;
      (1:1ØØ):   COUNT = COUNT + 1;
      (1Ø1:5ØØ): COUNT = COUNT + 2;
      (5Ø1:9ØØ): COUNT = COUNT + 3;
      END
   END
 TERM
```

As a result the following program is compiled:

```
START PROGRAM MAIN;
   BEGIN
   ITEM RESULT U;
   ITEM COUNT U;
   REF PROC RANDOM U;
      BEGIN
      END
   RESULT = RANDOM;
   COUNT = Ø;
   CASE RESULT;
      BEGIN
      ( DEFAULT ): ;
      (1:1ØØ):    COUNT = COUNT + 1;
      (1Ø1:5ØØ): COUNT = COUNT + 2;
      (5Ø1:9ØØ): COUNT = COUNT + 3;
      END
   END
TERM
```

17.3 LISTING-DIRECTIVES

The listing-directives are used to provide the compiler with information about which parts of the source listing are to printed and where page ejects are desired. Three listing-directives are defined:

```
!LIST;

!NOLIST ;

!EJECT ;
```

If no listing-directives are given, the compiler prints a listing of the source program, inserting page breaks in an implementation dependent manner. The !NOLIST directive tells the compiler to suppress the listing of the source program. The !LIST directive tells the compiler to resume listing the source program. The !EJECT directive tells the compiler to insert a page break.

17.3.1 Placement

The listing-directives can be placed anywhere a directive can be given.

17.4 INITIALIZATION-DIRECTIVE

The initialization-directive is used to set all static data that is not initialized by a preset to zero bits. The form of the initialization directive is:

```
!INITIALIZE ;
```

The effect of the initialization directive extends from the point at which it is given to the end of the current scope.

17.4.1 Placement

The initialize-directive can only be given before a declaration. However, it cannot be given before a declaration that is within a table or a block. Further, it cannot be given before a subroutine-declaration.

17.4.2 Example

Consider the following program fragment:

```
!INITIALIZE;
ITEM COUNT U;
TABLE SPECS(100);
  BEGIN
  ITEM LENGTH U;
  ITEM WIDTH U = 101(5);
  ITEM HEIGHT U;
  END
```

The initialize-directive causes COUNT and the 101 instances of the items LENGTH and HEIGHT of the table SPECS to be initialized. The 101 instances of the item WIDTH are preset to 5 and are, therefore, not affected by the initialize-directive.

17.5 ALLOCATION-ORDER-DIRECTIVE

The allocation-order-directive instructs the compiler to allocate storage for the data objects in a block or table in same order as their declarations are given. If an allocation-order-directive is not given, the compiler can rearrange the physical storage layout of the data objects within a block or table to provide for better access or better use of storage.

The form of the allocation-order-directive is:

```
!ORDER ;
```

17.5.1 Placement

The allocation-order-directive can be given only as the first entity in a block-description or entry-description. The effect of an allocation-order-directive extends from the point at which it is given to the end of the current block or table.

An allocation-order-directive can also be given in a type-declaration. When the type-name declared in this way is used, the allocation-order-directive applies to the object being declared.

17.5.2 Example

Supppose you have the following table:

```
TABLE PARTS(1000) D;
   BEGIN
   ITEM ID U 5;
   ITEM NUMBER U;
   ITEM FLAG B;
   END
```

The letter D in the table-attributes indicates dense packing. Dense packing is an advanced topic described in Chapter 19.

If the compiler is allowed to change the order of allocation, it can allocate ID and FLAG in a single word and conserve storage. (Not all compilers perform this sort of rearrangement.) However, if you want to be certain that no rearrangement occurs, you can include an allocation-order-directive as follows:

```
TABLE PARTS(1000) D;
   BEGIN
   !ORDER;
   ITEM ID U 5;
   ITEM NUMBER U;
   ITEM FLAG B;
   END
```

17.6 EVALUATION-ORDER-DIRECTIVES

The evaluation-order-directives are used to indicate whether or not the compiler can rearrange computations within a formula.

The evaluation-order-directives are:

!LEFTRIGHT ;

!REARRANGE ;

The !LEFTRIGHT directive tells the compiler that it must evaluate operators at the same precedence level from left to right within a formula. The !REARRANGE directive tells the compiler that it can evaluate operators at the same precedence level in any order when such a rearrangement produces more efficient code. Evaluation order is of course, constrained by parentheses.

If no directive is given, the compiler assumes that it can rearrange the evaluation order of operators of the same precedence.

17.6.1 Placement

These directives can be placed anywhere a directive can be given.

The effect of an evaluation-order-directive extends from the point at which it is given to the end of the scope or to the next evaluation-order-directive, whichever comes first.

17.6.2 Example

Suppose you have the following formula:

```
HEIGHT*LENGTH*WIDTH
```

If no evaluation-order-directive is given, the compiler can rearrange the formula as follows:

```
LENGTH*HEIGHT*WIDTH
```

Or it can rearrange in any other way to produce efficient code. However, if the !LEFTRIGHT directive is in effect, the compiler must first multiply HEIGHT times LENGTH and then multiply the result by WIDTH.

17.7 INTERFERENCE-DIRECTIVE

The interference-directive is used to inform the compiler that it cannot assume that the storage for the given names is distinct. The form of the interference-directive is:

```
!INTERFERENCE  data-name : data-name ,...  ;
```

The interference-directive indicates that the storage for the first data-name is not necessarily distinct from the storage for the list of data names following the colon.

The names given in the interference-directive must have been previously declared.

If an interference-directive is not given, the compiler assumes that distinct data names refer to distinct storage and makes optimizations based on that assumption.

The compiler is aware of storage that overlaps because of language features that allow overlaying. These language features, specified tables and overlay declarations, are described in Chapter 19 on "Advanced Topics". However, there are cases in which the compiler is not aware of overlaps and for these cases an interference directive must be given. For example, if two data objects are assigned the same absolute address in different overlay-declarations, an interference-directive should be used to warn the compiler.

17.7.1 Placement

An interference-directive can be given only before a declaration.

17.7.2 Example

As an example of the use of the interference-directive, consider the following:

```
TABLE PARTS(10);
   ITEM PARTNO U;
ITEM SIZE F;
ITEM ID F;
OVERLAY POS(3310) PARTS;
OVERLAY POS(3314) SIZE;
!INTERFERENCE PARTS : SIZE, ID;
```

This directive informs the compiler that it should not assume that the storage for PARTS is distinct from the storage for SIZE and ID.

17.8 REDUCIBLE-DIRECTIVE

The reducible-directive is used to allow additional optimizations of function calls. The form is:

```
!REDUCIBLE ;
```

A reducible function is one that has the following characteristics:

- All calls with identically valued actual parameters result in identical function values and output parameter values.
- The only data that is modified by the function call is that data declared within the function.

The compiler can, in some cases, detect the existence of common calls on a reducible function, save the values produced by the first call, delete subsequent calls and use the values produced by the first call.

17.8.1 Placement

A reducible-directive is given following the semicolon of the function heading. A reducible function must have the reducible-directive in its definition and all its declarations.

17.8.2 Example

Trigonometric functions are good examples of reducible functions. SIN(ANGLE) always produces the same result for the same value of ANGLE and the function has no side effects.

17.9 REGISTER-DIRECTIVES

Register-directives are used to affect target-machine register allocation. Three register-directives are defined, namely:

```
!BASE data-name register-number;

!ISBASE data-name register-number;

!DROP register-number ;
```

Register-number is an integer literal that specifies the register in a target-machine-dependent way.

Both the !BASE and !ISBASE directives cause the compiler to dedicate the register to the value it currently contains. The !BASE directive instructs the compiler to load the specified register with the address of the given data-name. The !ISBASE directive instructs the compiler to assume that the specified register contains the address of the data object.

The !DROP directive frees the specified register for other use by the compiler.

Register allocation is not meaningful for all machines. Register-directives are ignored for machines that do not use registers.

17.9.1 Placement

The register-directives can be given anywhere a directive can be given.

17.1Ø LINKAGE-DIRECTIVE

The linkage-directive is used to identify a subroutine that does not obey standard JOVIAL (J73) linkage conventions. The form of the linkage-directive is:

```
!LINKAGE symbol ... ;
```

Symbol in a linkage-directive is a string that specifies the implementation-dependent linkage type to be used in linking the procedure.

17.1Ø.1 Placement

A linkage-directive can be given only in a subroutine declaration or a subroutine definition. It is given there between the heading and the declaration of the formal parameters.

17.1Ø.2 Example

Suppose you want the following subroutine to have non-standard linkage. You can write the following subroutine-declaration:

```
PROC INTERFACE(CHANNEL:UNIT);
!LINKAGE ASSEMBLY;
BEGIN
ITEM CHANNEL U;
ITEM UNIT U;
END
```

17.11 TRACE-DIRECTIVES

The trace-directives are used to follow program execution and monitor data assignments. The trace-directive has one of the following forms:

```
!TRACE ( control ) name ,... ;

!TRACE name ,... ;
```

The first form of the trace-directive is a conditional trace. It causes tracing only if control, which is a boolean formula, is TRUE. The second form is an unconditional trace.

The names given in the trace-directive are the names to be traced. A name can be a statement name, a subroutine name, or a data name.

- For a statement name, the trace notes each time the associated statement is executed.
- For a subroutine name, the trace notes each call on the subroutine. If the subroutine name given is the subroutine that contains the trace-directive, the trace notes both entry to and exit from the subroutine.
- For a data name, the trace notes any modification of the value of the data object. The new value is included in the trace printout. If the data name is a table, the trace notes any modification of a table item, a table entry, or the entire table. If the data name is a block, the trace notes modification of any enclosed object.

Data names given in the control or as names to be traced must be declared previously. Statement or subroutine names can be declared later.

17.11.1 Placement

A trace-directive can be given only before a statement. It applies from the point at which it is given to the end of the scope.

Chapter 18

DEFINE CAPABILITY

The define capability is used to associate a name with a string of JOVIAL (J73) text. When the name is used in a program, the compiler substitutes the associated string for the name.

The following sections describe the declaration and use of define-names.

18.1 DEFINE-DECLARATION

The simplest form of the define-declaration simply associates a string with a name, as follows:

```
DEFINE define-name  "define-string" ;
```

The define-string is any sequence of JOVIAL (J73) characters.

Suppose you want to define a name MAXSIZE as the quotient of the implementation parameters MAXBITS over BITSINWORD. You can use a define-declaration as follows:

```
DEFINE MAXSIZE "MAXBITS/BITSINWORD";
```

This declaration declares the define-name MAXSIZE and associates with it the define-string "MAXBITS/BITSINWORD".

A define-declaration can also contain parameters. The form with parameters is:

```
DEFINE define-name ( define-formal ,... ) "define-string" ;
```

The character sequence ",..." indicates that one or more define-formals can be given separated by commas.

A define-formal is a single letter. Within the parenthesized parameter list, define-formals are indicated by that single letter. Within the define-string, define-formals are indicated by that letter preceded by an exclamation point. A define-formal receives its value from the corresponding define-actual given in a call on the define-name.

For example, to provide a convenient rotation for incrementation, you can define a name TALLY and associate it with the following string:

```
DEFINE TALLY(A) "!A = !A + 1" ;
```

The define-name TALLY has one define-formal, A, associated with it.

A define-declaration can also include a list-option, which describes how much information is to be given in the output listing. The general form of the define-declarations is:

```
DEFINE define-name [ ( define-formal ,... ) ] [ list-option ]

                "define-string" ;
```

The square brackets indicate that both the parenthesized list of define-formals and the list option are optional.

The parameters, define-string, and list-option are discussed in detail later in this chapter.

18.2 DEFINE-CALLS

A define-call directs the compiler to make a copy of the define-string associated with the define-name, replace the define-formals by the define-actuals in that copy, and replace the define-call by the resulting string. The form of the define-call is:

```
define-name [ ( define-actual ,... ) ]
```

The square brackets indicate that the parenthesized list of define-actuals is optional. The sequence ",..." indicates that if more than one define-actual is given, the define-actuals are separated by commas.

A define-call for a define-name that is declared without parameters is simply the define-name alone.

For example, a define-call for the define-name MAXSIZE, declared earlier in this chapter, is simply:

```
MAXSIZE
```

When the compiler sees MAXSIZE, it substitutes the associated define-string MAXBITS/BITSINWORD. For example, you can write:

```
IF SIZE < MAXSIZE;
   EXIT;
```

The compiler substitutes the define-string associated with MAXSIZE to get the following:

```
IF SIZE < MAXBITS/BITSINWORD;
   EXIT;
```

A define-call for a define-name that is declared with parameters can have a list of define-actuals. Define-actuals can be omitted, if a meaningful result is produced. Examples of define-calls with missing define-actuals are given later in this chapter.

For example, the define-name TALLY, declared earlier in this chapter, has one define-formal associated with it. Omitting the define-actual does not produce a meaningful result, so a define-call for TALLY must have one define-actual, as follows:

```
TALLY(COUNT);
```

In place of this define-call, the compiler uses a copy of the define-string associated with the name TALLY in which the define-formal A is replaced by the define-actual COUNT. That is, it supplies the following substitution:

```
COUNT = COUNT + 1;
```

18.2.1 Placement

The compiler only interprets a define-call that is a symbol within the program. It does not process the characters within comments and character literals. Therefore, a define-call in either of those places is not expanded.

18.3 THE DEFINE-STRING

The define-string can consist of any string of characters within the enclosing quotes. Since the quote and exclamation point characters have special meaning within a define-string, these characters must be doubled to be used as simple characters within a define-string.

Suppose you want to define a statement that includes a comment, as follows:

```
DEFINE ALERT "IF READY; ALARM; ""PHASE 1""";
```

The quotes enclosing the comment are doubled so that the compiler can interpret them as characters and not as delimiters of the define-string.

When you use ALERT in your program, the compiler substitutes the associated define-string, as follows:

```
IF READY; ALARM;  "PHASE 1"
```

18.3.1 Define-Calls in Define-Strings

A define-string can include define-calls. The compiler, in expanding a define-call, first makes a copy of the associated define-string, then substitutes the define-actuals for the define-formals, then examines the resulting string to see if it contains any define-calls. If it does, the compiler expands these define-calls in the same way. Expansion is complete when the resulting string cannot be processed further; that is, does not contain any more define-calls.

Suppose you have the following declarations:

```
DEFINE T1(A,B) "!A/!B**EXP";
DEFINE EXP  "2";
```

Now consider the use of the define-name T1:

```
XCOORD = T1(YCOORD,5);
```

The compiler first expands T1 to get the following:

```
XCOORD = YCOORD/5**EXP;
```

It then expands EXP and substitutes the resulting string in the assignment statement as follows:

```
XCOORD = YCOORD/5**2;
```

Suppose that two different define-declarations exist for EXP in different scopes, as follows:

```
PROC CALCULATE;
  BEGIN
  DEFINE T1(A,B) "!A/!B**EXP";

  ( declarations and statements )

  PROC COMP1;
     BEGIN
     DEFINE EXP "2";

     ( declarations and statements )

     XCOORD = T1(YCOORD,5);

     ( statements )

     END
  PROC COMP2;
     BEGIN
     DEFINE EXP "5";

     ( declarations and statements )

     XCOORD = T1(YCOORD,5);

     ( statements )

     END
  END
```

The define-call on T1 in the procedure COMP1 is expanded as follows:

```
YCOORD/5**2
```

The define-call on T1 in the procedure COMP2 is expanded as follows:

```
YCOORD/5**5
```

18.3.2 Comments in Define-Declarations

Comments can appear anywhere in the language except between the define-name and the define-string. The compiler interprets the first quoted string it finds following the define-name as the define-string.

Suppose you write the following:

```
DEFINE COEF "(2*FACTORIAL(NEXT)-1)" "BEST APPROXIMATION";
```

The compiler assumes that COEF is followed by a define-string and then a comment. Suppose you use COEF as follows:

```
TERM = COEF * LAST;
```

The compiler substitutes the define-string as follows:

```
TERM = (2*FACTORIAL(NEXT)-1) * LAST;
```

18.4 DEFINE PARAMETERS

The define-actuals given in the define-call are associated with the define-formals given in the define-declaration. The first (leftmost) define-actual in the define-call is associated with the first (leftmost) define-formal in the declaration; the second define-actual with the second define-formal, and so on.

18.4.1 Define-Actuals

A define-actual can be any sequence of characters. It can include the comma character and the parentheses characters. The rule for delimiting a define-actual is to use the characters up to but not including one of the following:

1. The first right parenthesis not balanced by a left parenthesis that is part of the define-actual.

2. The first comma that is not within a pair of balanced parentheses within the define-actual.

Quotes can be used around define-actuals that must include an unbalanced right parenthesis or a comma that is not within parentheses. Two quote characters must be used to represent a single quote character within a define-actual that is enclosed in quotes.

The following list gives some define-actuals for the associated define-call.

Define-Call	Define-Actuals No.	Define-Actuals Value
TASK(A,B,C)	1	A
	2	B
	3	C
TASK(A(B,)C)	1	A(B,)C
TASK("A,","B,",C)	1	A,
	2	B,
	3	C
TASK((A,B,C))	1	(A,B,C)
TASK("AB""C")	1	AB"C

18.4.2 Missing Define-Actuals

If a define-actual is not given for a define-formal, a null string is substituted for the define-formal. Define-actuals can be omitted at the end of the parameter list. Within the parameter list, adjacent commas indicate the omission of a define-actual.

Suppose you have the following define-declaration:

```
DEFINE COMPUTE(A,Z) "VELOCITY = RATE!A/DISTANCE!Z;"
```

The following define-calls produce the indicated results:

Define-Call	Result
COMPUTE(Q1,X2)	VELOCITY = RATEQ1/DISTANCEX2;
COMPUTE(1)	VELOCITY = RATE1/DISTANCE;
COMPUTE(,OBS)	VELOCITY = RATE/DISTANCEOBS;
COMPUTE()	VELOCITY = RATE/DISTANCE;

18.5 GENERATED NAMES

A define-declaration can be used to generate names by the placement of the define-formals, as shown in the declaration of COMPUTE.

As another example, suppose you have the following define-declaration:

```
DEFINE NEWSYMBOL(A) "XYZ!A";
```

You can use the define-call as a variable, as follows:

```
NEWSYMBOL(1) = 0;
```

The generated name XYZ1 is substituted in this statement to produce:

```
XYZ1 = 0;
```

A define-call must not be used, however, as the name being declared in a declaration. Generated names must be declared previously in the conventional way.

Further, define-calls cannot be used to create a new symbol by virtue of concatenating the define-call with the surrounding text. Suppose you have the following define-declaration:

```
DEFINE STAR "*";
```

Now, suppose you use that define-name in a statement as follows:

```
LENGTH = OBSERVED STAR* 2;
```

The compiler expands the define-call STAR, but does not interpret the result as an exponentiation operator. It treats the statement as having two multiplication operators and rejects it as syntactically incorrect.

The define-name STAR can be used in a valid way as follows:

```
LENGTH = OBSERVED STAR CORRECTION;
```

The compiler expands the define-name STAR to create the following valid statement:

```
LENGTH = OBSERVED * CORRECTION;
```

18.5.1 Context

The expansion of a define-call must produce a meaningful result.

Suppose you have the following define-declaration:

```
DEFINE SQUARE(A) " !A = !A**2;";
```

The define-actual in this case must be a variable to produce a valid statement.

18.6 DEFINE-CALLS IN DEFINE-ACTUALS

A define-call can be included in a define-actual. As described earlier in this chapter, the compiler expands a define-call by making a copy of the associated define-string and then substituting the define-actuals for the define-formals. If the resulting string contains any define-calls, the compiler expands them in the same way.

Thus a define-call that is part of a define-actual is expanded if, after the substitution of the define-actual, the define-call is a symbol and not part of a symbol.

Suppose you have the following define-declarations:

```
DEFINE DF1(A)  "!A1 = !A;";
DEFINE FUNCTION "SIN";
```

Consider the following define-call:

```
DF1(FUNCTION)
```

The compiler copies the define-string associated with DF1 and substitutes the define-actual FUNCTION for the define-formal A to produce the following string:

```
FUNCTION1 = FUNCTION;
```

The first instance of FUNCTION is part of a symbol and, therefore, the compiler does not recognize it as a define-call. The second instance of FUNCTION is a define-call and is expanded. The result of that expansion is:

```
FUNCTION1 = SIN;
```

The text is now fully expanded.

18.7 THE LIST OPTION

The list option lets you specify whether you want to see the define-string in your program, or the define-call, or both. The list options are:

LISTEXP	Include the expanded define-string in the listing in place of the define-call.
LISTINV	Use the define-call in the listing and do not include the expansion.
LISTBOTH	Include both the define-call and the resulting expansion in the listing.

The exact format of the output listing is implementation dependent.

Chapter 19

ADVANCED TOPICS

This chapter considers some advanced topics. It begins by describing the different ways in which you can lay out a JOVIAL (J73) table in storage. It next describes the overlay-declaration, which lets you determine the data objects that can share storage and lets you allocate data at specific machine addresses. It then considers the way in which you can determine the size and representation of status constants. It concludes with a discussion of DEF-block-instantiations.

19.1 JOVIAL (J73) TABLES

A JOVIAL (J73) table can be either an ordinary table or a specified table. An ordinary table is one in which the compiler determines the storage layout subject to information supplied in the declaration about the structure and packing of the table. A specified table is one in which the declaration completely describes the storage layout of every item.

The following sections describe these two types of tables in detail.

19.2 ORDINARY TABLES

The declaration of ordinary tables was described in Chapter 7. This section considers two additional specifiers that can be included in the table-declaration for an ordinary table.

These specifiers provide information about the structure and packing of the table. The structure-spec describes the structure of the table in memory (serial or parallel) or the number of entries to be packed per word (tight structure). The packing-spec describes the way in which items within a word are packed.

19.2.1 Packing

Table packing refers to the allocation of items within an entry to words of storage. If a table entry contains more than one item, the way in which the items of the entry are packed can be specified by giving a packing-spec:

The packing-spec can be given as part of the table declaration, as follows:

```
     TABLE  table-name [ ( dimensions ) ]

                                  [  packing-spec ] ;

        entry-description
```

The square brackets indicate that the parenthesized dimensions, the structure-spec and the packing-spec are all optional.

A packing-spec can also be given for any item in the table, as follows:

```
     ITEM  item-name  item-description   [ packing-spec ] ;
```

If the packing-spec is given in the table-attributes, it applies to the entire table. That is, all items are packed according to that packing-spec except those items that have a packing-spec in their declaration.

The packing-spec is one of the following:

- N No packing occurs. Each item begins in a new word.
- M Medium packing occurs. The amount of packing depends on the implementation.
- D Dense packing occurs. The compiler packs as many items as possible within a word, making use of all available bits within the word. However, items that occupy one word or more are always allocated at a word boundary and the bytes of a character item are always aligned on a byte boundary. Further, if the structure of the table is parallel, no item is allocated so that it crosses a word boundary.

If a packing-spec is not given, the compiler assumes N (no packing) for serial and parallel tables and D (dense packing) for tables with tight structure. Table structure is described in the next section.

Consider the following declaration:

```
TABLE TRACK(1:100);
   BEGIN
   ITEM DIST U 5;
   ITEM SB B 3;
   ITEM ANGLE S 10;
   END
```

Suppose that BITSINWORD is 16. Since no structure-spec or packing-spec is given, the compiler assumes a serial table with no packing and allocates each item to a separate word. It can be diagrammed as follows:

```
TRACK

+---------------------+
| DIST(1)             |
|---------------------|
| SB(1)               |        TRACK(1)
|---------------------|
| ANGLE(1)            |
|---------------------|

    ...

+---------------------+
| DIST(100)           |
|---------------------|
| SB(100)             |        TRACK(100)
|---------------------|
| ANGLE(100)          |
+---------------------+
```

The table TRACK, in this case, requires 300 words of storage.

Now consider a table declaration for the same table that includes a packing-spec of D:

```
TABLE TRACK(1:100) D;
   BEGIN
   ITEM DIST U 5;
   ITEM SB B 3;
   ITEM ANGLE S 10;
   END
```

Again asssuming that BITSINWORD is 16, the compiler packs as many items of the entry as possible within a word. The total number of bits required is 19 and thus the compiler uses two words for each entry. The exact layout of the items within those words is implementation dependent. It can be diagrammed as follows:

```
TRACK                                      BITSINWORD is 16

+--------------------+
| DIST(1)    |  SB(1) |
|--------------------|
| ANGLE(1)           |                     TRACK(1)
|--------------------|

      ...

+--------------------+
| DIST(100) | SB(100) |
|--------------------|
| ANGLE(100)         |                     TRACK(100)
+--------------------+
```

The table, in this case, requires 200 words of storage.

If BITSINWORD is 32, then the compiler is able to pack all three items of an entry into a single word. That layout can be diagrammed as follows:

```
TRACK                                            BITSINWORD is 32

+-----------------------------------+
|  DIST(1)   |  SB(1)   | ANGLE(1)   |  TRACK(1)
|-----------------------------------|
    ...

|-----------------------------------|
|  DIST(100)|  SB(100)| ANGLE(100)  |  TRACK(100)
+-----------------------------------+
```

The table, in this case, needs only 100 words of storage.

Now, consider a table declaration for the same table that includes a D packing-spec in the table-attributes and an N packing-spec in the item-declaration of SB:

```
TABLE TRACK (1:100) D;
   BEGIN
   ITEM DIST U 5;
   ITEM SB B 3 N;
   ITEM ANGLE S 10;
   END
```

The packing-spec for the table indicates dense packing, but the packing-spec for item SB indicates no packing. All other items in the table can be packed densely, but item SB must occupy a word by itself.

If the given implementation reorders items and if an !ORDER directive is not in effect, it can pack DIST and ANGLE in one word and allocate SB in another word. Such a layout can be diagrammed as follows:

```
TRACK                                BITSINWORD is 16

+------------------------+
| DIST(1)    | ANGLE(1)  |
|------------------------|
| SB(1)                  |        TRACK(1)
|------------------------|

    ...

+------------------------+
| DIST(100) | ANGLE(100) |
|------------------------|
| SB(100)                |        TRACK(100)
+------------------------+
```

If the implementation does not perform reordering or if an !ORDER directive is in effect, then the items each occupy a word and the table requires 300 words of storage.

Consider another case in which the table does not have a packing-spec and therefore N (no packing) is assumed. Several items within the table, however, have packing-specs of D, as follows:

```
TABLE SUPERTRACK(1ØØ);
   BEGIN
   ITEM DIST U 5;
   ITEM SB B 3 D;
   ITEM ANGLE S 1Ø;
   ITEM MASK1 B 4 D
   ITEM MASK2 B 2 D;
   END
```

This declaration effectively directs the compiler to allocate a separate word for DIST and a separate word for ANGLE and to pack MASK1 and MASK2 within a single word.

If the implementation of the compiler performs reordering and if the !ORDER directive is not present, it can pack SB, MASK1, and MASK2 in the same word.

19.2.2 Structure

Table structure refers to the way in which the entries of a table are laid out in memory. JOVIAL (J73) permits two fundamental types of structure, serial and parallel.

A serial table can be structured as either an ordinary serial table, in which the compiler starts each entry in a new word, or a tight serial table, in which the compiler packs as many entries as possible within a word.

The structure-spec is given in the table declaration following the parenthesized dimension-list.

```
TABLE name ( dimensions ) [ structure-spec ]

                    [ packing-spec ] ;

     entry-description
```

The square brackets indicate that the structure-spec is optional. Although the parenthesized dimension list is optional in a table-declaration, a structure-spec is meaningful only when the table is dimensioned.

Structure-spec is one of the following:

    PARALLEL

    T [entry-size]

The square brackets indicate that entry-size is optional.

The letter T indicates a tight structure. Entry-size is a compile-time-integer-formula that gives the number of bits for each entry. If entry-size is not given, the compiler uses the minimum number of bits necessary to represent the entry for entry-size. If no structure-spec is given, the compiler assumes that the table is an ordinary serial table.

19.2.2.1 Serial Structure

The compiler lays out a serial table by taking the first word of the first entry, followed by the second word of the first entry, and so on.

19.2.2.2 Parallel Structure

The compiler lays out a parallel table by taking the first word (word Ø) of the first entry followed by the first word of the second entry and so on to the first word of the last entry, then the second word (word 1) of the first entry, the second word of the second entry, and so on.

An important restriction on the use of parallel tables is that PARALLEL structure can be specified only for a table in which none of the items of an entry occupy more than one word. A table is layed out in a parallel structure on a word-by-word basis, even for packed tables.

19.2.2.3 Example of Serial vs. Parallel Structure

Consider the following two table declarations:

```
TABLE RACE1(100);          TABLE RACE2(100) PARALLEL;
   BEGIN                      BEGIN
   ITEM SPEED U;              ITEM SPEED U;
   ITEM DISTANCE S;           ITEM DISTANCE S;
   END                        END
```

These declarations are the same except that table RACE1 is specified (by default) as having a serial structure and table RACE2 is specified as having a parallel structure.

The compiler lays out these tables a follows:

```
     RACE1 (Serial)             RACE2 (Parallel)

  +-----------------+        +-----------------+
  |  SPEED(0)       |        |  SPEED(0)       |
  |-----------------|        |-----------------|
  | DISTANCE(0)     |        |  SPEED(1)       |
  |-----------------|        |-----------------|
  | SPEED(1)        |        |  SPEED(2)       |
  |-----------------|        |-----------------|
  | DISTANCE(1)     |        |  SPEED(3)       |
  |-----------------|        |-----------------|

     ...                        ...

  |-----------------|        |-----------------|
  |  SPEED(100)     |        |  DISTANCE(99)   |
  |-----------------|        |-----------------|
  |  DISTANCE(100)  |        |  DISTANCE(100)  |
  +-----------------+        +-----------------+
```

The serial organization of RACE1 is appropriate if your program uses SPEED and DISTANCE together. If your program processes an item in the first word of each entry, then later an item in the second word of each entry, you can localize addressing by creating a parallel table. Such localization may produce a more efficient program, but the effect of localization depends on the length of the table, the machine's method of addressing and many other factors.

19.2.2.4 Tight Structure

If the entries of a table each occupy less than one word, the entries can be packed. Entries in tight tables can have more than one item, but the entire entry cannot exceed a word in length. In fact, in order for entry packing to occur, the entry cannot exceed half the word length.

If a tight structure is not specified, the compiler begins each entry in a new word. Consider the following declaration:

```
TABLE ATTENDANCE(1:1000);
   ITEM COUNT U 5;
```

This declaration causes the compiler to create a serial table. This table can be diagrammed as follows:

```
ATTENDANCE

+------------------------+
| COUNT(1)               |      ATTENDANCE(1)
|------------------------|

      ...

|------------------------|
| COUNT(1000)            |
+------------------------+
```

Each entry in the table occupies one word and thus the table is 1000 words long.

Now consider a declaration of the same table with tight structure:

```
    TABLE ATTENDANCE(1:1000) T;
       ITEM COUNT U 5;
```

The T structure-spec directs the compiler to pack as many entries as possible within a word. Suppose BITSINWORD is 16. The compiler can pack three entries per word. The table can be diagrammed as follows:

```
ATTENDANCE                              BITSINWORD=16

+------------------------------------+
|COUNT(1)    |COUNT(2)    |COUNT(3)   |
|------------------------------------|
|COUNT(4)    |COUNT(5)    |COUNT(6)   |
|------------------------------------|

      ...
|------------------------------------|
|COUNT(997)  |COUNT(998)  |COUNT(999) |
|------------------------------------|
|COUNT(1000)                          |
+------------------------------------+
```

The table now occupies 334 words.

Entry-size allows the packing to be given such that entries begin on addressable units. For example, consider a declaration of the same table with a specified entry-size:

```
TABLE ATTENDANCE(1:1000) T 8;
   ITEM COUNT U 5;
```

If BITSINWORD is 16, the compiler can pack two entries per word since it must use 8 bits for each entry. If BITSINBYTE is 8, each entry begins on a byte boundary. This table can be diagrammed as follows:

```
       ATTENDANCE                            BITSINWORD is 16

  +------------------------------+
  | COUNT(1)   xxx|COUNT(2)   xxx|
  |------------------------------|
  | COUNT(3)   xxx|COUNT(4)   xxx|
  +------------------------------|        x indicates an unused bit

         ...

  |------------------------------|
  |COUNT(999) xxx|COUNT(1000)xxx|
  +------------------------------+
```

This table occupies 500 words.

The default packing-spec for a tight serial table is D (dense). A tight table uses the minimum necessary storage. For example, suppose you declare the following table:

```
TABLE GRID (20) T;
   BEGIN
   ITEM XCOORD U 5;
   ITEM YCOORD U 5;
   END
```

The compiler uses dense packing. Since entry-size is not given, the compiler uses the minimum number of bits necessary for an entry - in this case, 10 bits. If BITSINWORD is 32, the compiler can then pack three entries per word.

19.2.3 Conversion and Packed Items

When an item is given in a packed table, the implemented precision is the same as the declared precision. Thus, an assignment to an item in a packed table can result in loss of significant digits in some cases.

For example, automatic conversion of a fixed data object does not change the numeric value of the data object except when the implemented precision of the result value is less than the implemented precision of the value being converted. In this case, rounding or truncation occurs with respect to the implemented precision of the converted value. This situation occurs only when assigning to a packed fixed table item. The round-or-truncate attribute of the table item determines whether the assigned value is rounded or truncated.

For example, suppose you have the following declarations:

```
TABLE FACTORS (1:100) D;
   BEGIN
   ITEM FIRST A 2,4;
   ITEM SECOND A 2,4;
   ITEM LAST  A 2,4;
   END
ITEM TEMP A 2,4;
```

The implemented precision of TEMP may be greater than the nominal precision given by the scale and fraction. The precision of FIRST, however, is 6 bits as indicated by the scale and fraction. Assigning TEMP to FIRST(I) thus probably involves rounding or truncating TEMP.

19.3 SPECIFIED TABLES

A specified table-declaration contains information about the position of each item of each entry.

19.3.1 Specified Table Type Declarations

A specified table can be used in any context in which an ordinary table can be used. In particular, it can be used in a type-declaration to create a type for a table with a particular layout.

A specified table has the same general form as an ordinary table, namely:

```
      TABLE  table-name  table-attributes ;

         entry-description
```

The specified table-kind is given in the table-attributes instead of a packing-spec, as follows:

```
      [ ( dimensions ) ] [ structure-spec ] [ table-kind ]
```

The table-kind indicates whether the table has fixed-length entries or variable-length entries. The forms are:

```
     W entry-size

     V
```

The W indicates that the table has fixed-length entries. The V indicates that the table has variable-length entries. Entry-size is an integer compile-time-formula that gives the number of words each entry occupies for a fixed length entry table.

The two kinds of specified table are considered in detail later in this chapter.

The position of each item in a specified table entry is given by a POS clause following the each item-description in the table, as follows:

```
     ITEM item-name item-description

          POS ( startbit, startword ) ;
```

Startbit and startword are integer compile-time-formulas. The first bit of a word is numbered Ø and the first word of an entry is numbered Ø.

Item positioning must take into account the number of bits in a word. An item that occupies one word or less must not be positioned so that it crosses a word boundary.

19.3.2 Tables with Fixed-Length Entries

A specified table with fixed-length entries is indicated by the specified-table-kind W followed by the entry size. A specified table with fixed-length entries can contain information about the structure and initial values. The form is:

```
TABLE table-name ( dimensions ) [ structure-spec ]

                    W entry-size  [ table-preset ] ;

        BEGIN
        ITEM  item-name item-description
              POS ( startbit , startword )  [ table-preset ] ;
        ...

        END
```

Suppose you need a table layout that corresponds to the format of a particular peripheral device. This format consists of two words per entry. Each word contains an unsigned, ten-bit integer left justified in that word. You can write the following table declaration:

```
TABLE DEVICE (5) W 2;
   BEGIN
   ITEM CHANNEL1 U 1Ø POS(Ø,Ø);
   ITEM CHANNEL2 U 1Ø POS(Ø,1);
   END
```

The table is a fixed-length specified table containing six entries. Each entry occupies two words. The first word contains the item CHANNEL1 in bits Ø through 9. The second word contains the item CHANNEL2 in bits Ø through 9.

19.3.2.1 The * Character

Every item in a specified table must be positioned. The asterisk character "*" can be used for startbit to indicate that the item should occupy the same amount of storage and be aligned in the same way as if it were allocated outside the specified table. In this way, the item can be accessed efficiently.

For example, suppose you use a specified table in the following way:

```
TABLE SURVEY(10) W 5;
   BEGIN
   ITEM FLAG B 3 POS(15,0);
   ITEM HISTORY B 10 POS(0,0);
   ITEM CASE1 U POS(*,1);
   ITEM CASE2 U POS(*,2);
   END
```

The items FLAG and HISTORY are positioned as indicated. The items CASE1 and CASE2 are positioned for efficient usage.

19.3.2.2 Overlays

The values of startbit and startword can be selected to overlay data. For example, consider the following declaration:

```
TABLE PERSONNEL(1000) W 3;
   BEGIN
   ITEM FLAG B 3 POS(15,0);
   ITEM NAME C 10 POS(0,1);
   ITEM RANK C 2 POS(0,6);
   ITEM ID C 4 POS(0,1);
   ITEM RATING C 2 POS(0,3);
   END
```

The items ID and RATING, in this declaration, overlay the item NAME.

19.3.2.3 Presets

If a table-preset is given in the table-attributes, then none of the item-declarations within the entry-description can have table-presets. If two items overlap, only one item can be given a preset.

Suppose you have the following table-declaration:

```
TABLE SPECS (100) W 2 = 2,4,,,6,8,,,10,12;
   BEGIN
   ITEM LENGTH U POS(0,0);
   ITEM HEIGHT U POS(0,1);
   ITEM HIPOINT U 8 POS(0,1);
   ITEM LOPOINT U 8 POS(8,1);
   END
```

The items are initialized in order and values are omitted for overlayed items. The first value 2 is used to set LENGTH(0), 4 is used to set HEIGHT(0). The omitted values prevent HIPOINT and LOPOINT from being initialized. The value 6 is used to set LENGTH(1), and so on.

19.3.2.4 Entry-Size

A specified table with fixed-length entries that does not have tight structure gets its entry size from the entry-size given following the W in the specified-table-kind. A specified table with fixed-length entries and tight structure gets its entry-size from the entry-size either given or assumed for the structure-spec. If a specified table has tight structure, entry-size must not be given as part of the specified-table-kind.

Suppose you declare the following table:

```
TABLE XR(9) T W;
   BEGIN
   ITEM READY B POS(0,0);
   ITEM STATBIT U 5 POS(1,0);
   END
```

Each entry contains a one-bit item and a five-bit item. Since the structure-spec does not give the number of bits in an entry, the compiler uses the minimum number of bits necessary to represent an entry, namely: six bits.

Assuming that BITSINWORD is 16, the items are allocated as follows:

Item	Word	Bits
READY(Ø)	Ø	Ø
STATBIT(Ø)	Ø	1-5
READY(1)	Ø	6
STATBIT(1)	Ø	7-11
READY(2)	1	Ø
STATBIT(2)	1	1-5

...

The starting bit in the position clause is assumed to be relative to the start of an entry. The item READY(1) is allocated at bit 6 of the first word. Its position, however, is bit Ø relative to the start of the entry. Observe that bits 12-15 of each word remain unused.

You may want to specify an entry size so that the entries of the table are allocated on addressable boundaries. For example, suppose BITSINWORD is 16 and BITSINBYTE is 8. You can write the following declaration to accomplish this:

```
TABLE XR(9) T 8 W;
  BEGIN
  ITEM READY B POS(Ø,Ø);
  ITEM STATBIT U 5 POS(Ø,1);
  END
```

The items are then allocated as follows:

Item	Word	Bits
READY(Ø)	Ø	Ø
STATBIT(Ø)	Ø	1-5
READY(1)	Ø	8
STATBIT(1)	Ø	9-13
READY(2)	1	Ø
STATBIT(2)	1	1-5

...

19.3.3 Tables with Variable-Length Entries

A table with variable-length entries in JOVIAL (J73) is indicated by the table-kind V. Such a table creates the illusion of being a variable length entry table, but it is, in fact, a table in which each entry is one word long.

A table with variable-length entries provides a way to save space by eliminating unnecessary items from entries.

A specified table with variable-length entries cannot contain a structure-spec or a table-preset. The form is simply:

```
TABLE table-name ( dimensions ) V;

   BEGIN
   ITEM item-name item-description POS ( startbit , word ) ;
   ...

   END
```

A physical entry in a table with variable-length entries is one word long. A logical entry in such a table can be, and usually is, composed of many items and may be several words long. The dimensions in a table with variable length entries determine the number of physical entries in the table. The number of logical entries depends on the way in which the table is built.

As a simple, but unrealistic, example of a table with variable-length entries, consider the following table-declaration:

```
TABLE ALTERNATOR(99) V;
   BEGIN
   ITEM A1 U POS(Ø,Ø);
   ITEM A2 U POS(Ø,1);
   ITEM B1 U POS(Ø,Ø);
   ITEM B2 U POS(Ø,1);
   ITEM B3 U POS(Ø,2);
   END
```

The table ALTERNATOR has two kinds of logical entry, a two word entry (consisting of A1 and A2) and a three word entry (consisting of B1, B2, and B3).

Suppose the table has alternating two and three word entries. The first logical entry consists of two words (A1 and A2) and begins at word 0. The second logical entry consists of three words (B1, B2, and B3) and begins at word 2. The third logical entry consists of two words and begins at word 5. And so on.

That is, the table looks as follows:

0	A1
1	A2
2	B1
3	B2
4	B3
5	A1
6	A2
	...
99	B3

To locate an item, the beginning of the logical entry is found and the position of the item within that entry is added to this base. The next entry is located by adding the number of items in the current entry to the base of the current entry.

Suppose you want to increment A2 in each two-word logical entry and B3 in each three-word logical entry. You can write:

```
TWO'WORD=TRUE;
FOR IX:Ø WHILE IX<99;
   IF TWO'WORD;
      BEGIN
      TWO'WORD=FALSE;
      A2(IX)=A2(IX)+1;
      IX = IX+2;
      END
   ELSE
      BEGIN
      TWO'WORD=TRUE;
      B3(IX)=B3(IX)+1;
      IX=IX+3;
      END
```

This fragment takes advantage of the fact that the logical entries alternate. It uses a switch TWO'WORD to determine which type of logical entry it is processing. This example is unrealistic because if the entries did alternate as shown, a five-word entry would be used. Normally, a logical entry must contain something within it to distinguish it.

Suppose you have a table that contains entries that are two, three and four words long, as follows:

Two-word-entry	Three-word-entry	Four-word-entry
ENTRY'SIZE	ENTRY'SIZE	ENTRY'SIZE
PART'NUMBER	PART'NUMBER	PART'NUMBER
	ON'HAND	ON'HAND
		DEFECTIVE

ENTRY'SIZE distinguishes the different kinds of logical entry. That is, A two-word entry contains ENTRY'SIZE with the value 2 and the number of the part (PART'NUMBER). A three-word entry contains ENTRY'SIZE with value 3, PART'NUMBER, and the number of units of that part currently available (ON'HAND). A four-word entry contains ENTRY'SIZE with the value 4, PART'NUMBER, ON'HAND, and the number of units of that part that have been found to be defective (DEFECTIVE).

You could use an ordinary table with four items in each entry for this table, but two words would then be wasted in entries that only need two words, and one word would be wasted in entries that only need three words.

You can, instead, use a table with variable-length entries, as follows:

```
TABLE PARTS (100) V;
  BEGIN
  ITEM ENTRY'SIZE U POS(0,0);
  ITEM PART'NUMBER  C 5 POS(0,1);
  ITEM ON'HAND U POS(0,2);
  ITEM DEFECTIVE U POS(0,3);
  END
```

Assuming a program has filled this table with entries, suppose you want to calculate the total number of defective items in the file. To do this, you look through the file and for each entry that contains a defective count, you add that count to a counter, COUNT.

You can locate those entries that have a DEFECTIVE item by the fact that the value of ENTRY'SIZE for an entry with a DEFECTIVE item is 4. The calculation is as follows:

```
COUNT = 0;
FOR I:0 THEN ENTRY'SIZE(I)+I WHILE I <100;
  IF ENTRY'SIZE(I)=4 THEN COUNT = COUNT + DEFECTIVE(I);
```

The loop statement uses ENTRY'SIZE to calculate the position of the next entry in the table. If that entry has four words, then it contains a defective unit count and that count is added to the counter COUNT.

19.4 THE OVERLAY DECLARATION

The overlay-declaration can be used for allocating several data objects in the same storage, for assigning data to a specified machine address, or for specifying the allocation order of a set of items.

The general form of the overlay-declaration is:

```
OVERLAY [ POS ( address ) ]

        overlay-expression ;
```

An overlay-expression is a sequence of one or more overlay-strings separated by colons, as follows:

```
overlay-string :...
```

An overlay-string consists of one or more overlay-elements, separated by commas, as follows:

```
overlay-element ,...
```

An overlay element is a name, a spacer, or a parenthesized overlay expression. The following sections consider these three types of overlay-element.

The data objects in an overlay-declaration can all be statically allocated or dynamically allocated, as long as all data objects have the same allocation permanence. An overlay-declaration must not be used to specify more than one physical location for any data object.

19.4.1 Data Names

The data names given in an overlay declaration must be previously declared. They can be item, table, or block names. But they cannot be the names of items within a table or items or tables within a block.

Further, an overlay-declaration can only name data that is declared without a REF-declaration and in the same scope as the overlay-declaration.

Consider the following declarations:

```
ITEM COUNT U;
ITEM TIME U;
ITEM MASK B 10;
ITEM RESULT F;
TABLE SPECIFICATIONS (99);
   BEGIN
   ITEM HEIGHT U;
   ITEM LENGTH U;
   ITEM WIDTH U;
   END
TABLE TEST(1:50);
   ITEM SUCCESS U;
```

Now consider the following overlay-declarations:

```
OVERLAY COUNT:TIME:RESULT;
OVERLAY SPECIFICATIONS:TEST,MASK;
```

The first overlay-declaration contains three overlay-strings. Each string contains one overlay-element. It specifies that the items COUNT, TIME, and RESULT are to share the same storage.

The second overlay declaration contains two overlay-strings. The first contains one overlay-element and the second contains two overlay-elements. It specifies that the table SPECIFICATIONS is to share the same storage as the table TEST and the item MASK. The table SPECIFICATIONS occupies 300 words. The first fifty words are shared with the table TEST and the fifty-first word is shared with the item MASK.

19.4.2 Spacers

An overlay element can also be a spacer, which indicates how many words to skip over when assigning storage. The form of the spacer is:

```
W words-to-skip
```

Words-to-skip is a compile-time integer formula that indicates how many words are to be skipped when allocating data in the overlay.

Suppose in the example given above, you want MASK to share the hundredth word with SPECIFICATIONS. You can write:

```
OVERLAY SPECIFICATIONS:TEST,W 49,MASK;
```

The table SPECIFICATIONS shares the first fifty words with TEST and the hundredth word with MASK. The words between TEST and MASK are not shared.

19.4.3 Nested Overlays

An overlay element can also be a parenthesized overlay element. For example, suppose you want TEST and COUNT to share the same storage as SPECIFICATIONS, and you want TIME to occupy the same storage as COUNT. You can write:

```
OVERLAY SPECIFICATIONS:TEST,(COUNT:TIME);
```

The table TEST shares the first fifty words of storage with SPECIFICATIONS and COUNT and TIME share the fifty-first word with SPECIFICATIONS and with each other.

19.4.4 Storage Sharing

When an overlay-declaration is used for storage sharing, it must have more than one overlay-string, as follows:

```
OVERLAY  overlay1 : overlay2 :... ;
```

The overlay-declaration asserts that the data objects in the first overlay occupy the same storage as the data objects in the subsequent overlays.

19.4.5 Allocating Absolute Data

The overlay declaration can also be used to allocate data at a specific machine address. The form of the overlay declaration for this case includes a positioner, as follows:

```
OVERLAY POS ( address ) overlay1 : ... ;
```

Address is an integer compile-time-formula that gives the address for a word.

Suppose you want to allocate COUNT at machine word 4500. You can write:

```
OVERLAY POS(4500) COUNT;
```

You can allocate a sequence of words, as follows:

```
OVERLAY POS(4500) COUNT, TIME, SPECIFICATIONS;
```

The item COUNT is allocated to word 4500, TIME to 4501, and SPECIFICATIONS to 4502 through 4802, assuming that 4500 is a decimal address.

You can also combine storage sharing with assigning absolute addresses. For example:

```
OVERLAY POS(4500) COUNT:TIME:TEST;
```

The items COUNT, TIME, and TEST are all allocated at machine address 4500.

An overlay-declaration with an absolute address cannot be given within a block.

19.4.6 Allocation Order

An overlay declaration can also be used to specify the order of allocation. Unlike the order-directive, which is used to specify allocation order within a table or block, the overlay-declaration is used to specify order in a more global way.

Suppose you want the items COUNT, TIME, and TEST to be allocated in that order. You can write:

```
OVERLAY COUNT, TIME, TEST;
```

This declarations assures the order of allocation for the three items given there.

19.4.7 Overlay-Declarations and Blocks

An overlay-declaration within a block must not reference names declared outside the block and an overlay-declaration outside a block must not reference names declared within the block.

Further, an overlay-declaration must not be given in a block if an order-directive is included in the block.

19.5 SPECIFIED STATUS LISTS

A status list can be given a specified representation. A specified representation associates given values with status constants.

The general form of a status type-description is:

```
STATUS  [ size ] ( status-group ,... )
```

The square brackets indicate that size is optional. Size is a compile-time-integer-formula that gives the number of bits to be used for the representation of the status-constants.

The characters ",..." indicate the one or more status-groups, separated by commas, can be given. Each status group has the form:

```
[ status-index ] status-constant ,...
```

If the status-index is not given, then the status-group has a default representation, as described earlier in Chapter 6. If the status-index is given, the status-group has a specified representation. Only status types with default representations can be used as dimensions in table declarations.

Suppose you want the status constants A through F to be represented as the values 10 through 15. You can write:

```
STATUS ( 10 V(A), V(B), V(C), V(D), V(E), V(F) )
```

If you want ALPHA to be represented as 2, BETA as 4, and GAMMA as 8, you can write:

```
STATUS ( 2 V(ALPHA), 4 V(BETA), 8 V(GAMMA) )
```

A status type-description can begin with a default list and continue with a specified list.

Suppose you want to associate the values Ø through 2 with the status constants CAR, VAN, and TRUCK, the values 8 through 9 with the status constants TRAIN and AIRPLANE, and the value 2Ø with the status constant SATELLITE. You can write:

```
STATUS ( V(CAR), V(VAN), V(TRUCK), 8 V(TRAIN), V(AIRPLANE),
         2Ø V(SATELLITE) )
```

No two status-constants in a given status list, however, can have the same representation.

19.6 DEF-BLOCK-INSTANTIATIONS

A def-block-instantiation is a special kind of external block declaration. A def-block-instantiation makes the name of the block external and allocates the block. The form is:

```
BLOCK INSTANCE block-name;
```

For each def-block-instantiation, a corresponding REF-declaration must be given, either in the same or in another module. The REF-declaration provides information about the components.

Appendix A

LANGUAGE SUMMARY

This appendix provides a syntactic summary for the JOVIAL (J73) language. The summary is divided up into a sequence of logical units. For each unit, the syntactic rules and a series of notes are given. The notes describe some of the most important facts and restrictions associated with the language contructs presented in the syntactic rules. At the end of the summary, and index to the syntactic terms is given.

A.1 INTRODUCTION

The following paragraphs define the notation used to present the syntax of JOVIAL (J73) and discuss the organization of this language summary.

A.1.1 Syntax Notation

A syntactic rule defines a syntactic name in terms of a string of syntactic terms. The syntactic terms can be terminals (such as: reserved words, separators, and the like), which are displayed in upper-case or syntactic names, which are displayed in lower-case.

Syntactic rules are displayed in boxes. The box is divided into a left-side and a right-side by a vertical line. On the left-side, the syntactic name being defined is given; on the right-side, the string that defines the name is given. For example, consider the following:

allocation-spec	STATIC

In the above rule, the syntactic name allocation-spec is defined to be the reserved word STATIC.

A.1.1.1 Concatenation

A concatenation is a sequence of two or more syntactic terms written one after the other. An example of a concatenation in a syntactic rule is:

block-preset	= block-preset-value

The above rule states that a block-preset is the character "=" followed by a block-preset-value.

A.1.1.2 Omission

If a construct is optional in a syntactic rule, it is enclosed in square brackets to indicate that it can be omitted.

An example of a rule with an omission is:

bit-type-description	B [bit-size]

This rule states that a bit-type-description is the letter B followed by an optional bit-size. That is, it can be either of the following:

B

B bit-size

A.1.1.3 Disjunction

A disjunction in a syntactic rule shows the set of possible choices in a syntactic definition. Curly braces are used to indicate disjunction. Within the curly braces, the choices are either separated from one another by vertical bars or are given on separate lines.

An example of a disjunction in which the choices are separated by vertical bars is:

ref-specification	REF { simple-ref | compound-ref }

This rule states that a ref-specification is the reserved word REF followed by either a simple-ref or a compound-ref. That is, it can be either of the following:

    REF simple-ref

    REF compound-ref

An example of a disjunction in which the choices are given on separate lines is:

status-constant	V ({ name letter reserved-word })

This rule states that a status-constant is the letter V followed by a parenthesized name, letter, or keyword. That is, it can be any of the following:

    V (name)

    V (letter)

    V (reserved-word)

A.1.1.4 Replication

A replication indicates that one or more repetitions of a construct can be given. The character sequence "..." is used to indicate replication. If the repetitions are separated by a punctuation character, then that character is given just before the three periods. For example, if the repetitions are separated by commas, the character sequence ",..." is used.

An example of a replication is:

positioner	POS (index ,...)

This rule states that a positioner is the reserved word POS followed by a left parenthesis followed by one or more indexes separated by commas followed by a right parenthesis. That is, it can be any of the following:

```
POS ( index )
POS ( index, index )
POS ( index, index, index )
    (and so on.)
```

If the construct to be repeated consists of more than one syntactic term, then curly braces are used to delimit the terms to be repeated. For example:

table-preset-value	{ [positioner] preset-option },...

This rule states that a table-preset-value is one or more pairs of optional positioner followed by preset-option separated by commas. That is, it can be any of the following:

```
[ positioner ] preset-option
[ positioner ] preset-option , [ positioner ] preset-option

 ( and so on )
```

A.1.2 Identical Definitions

If more than one syntactic name is defined by the same rule, a curly brace is used on the left-side of the box to indicate this fact. For example:

true-alternative false-alternative }	statement

This rule states that a true-alternative is a statement and a false-alternative is a statement.

A.1.3 Notes

The notes that follow a set of syntax rules list some important or hard-to-remember facts about the rules.

A.1.4 Syntax Index

The appendix is organized so that, wherever possible, syntactic terms that are used in a rule are defined on the same page. However, since this organization cannot always be achieved, a special index of syntactic terms is provided at the end of this appendix.

For example, consider the following rule:

item-declaration	ITEM item-name [STATIC] { type-description item-type-name } [item-preset] ;

Item-name and type-description are defined on the same page, but to conveniently locate the definitions for item-type-name and item-preset, you need to use the syntax index.

A.2 SYNTACTIC SUMMARY

module	{ main-program-module compool-module procedure-module }
main-program-module	START [dir ...] PROGRAM name ; [dir ...] program-body [[DEF] subroutine-definition ...] [dir ...] TERM
compool-module	START [dir ...] COMPOOL name ; [declaration ...] [dir ...] TERM
procedure-module	START [declaration ...] [[DEF] subroutine-definition ...] [dir ...] TERM

Notes:

1. A program is a set of modules. The modules are not necessarily all in the same file; details depend on the implementation.

2. A program must have exactly one main-program-module. It can have any number (perhaps none) of compool-modules or procedure modules.

3. A compool-module must not contain an inline-declaration or an item, table, block, statement-name, or subroutine declaration that does not begin with a DEF.

dir	!COMPOOL [compool-list] ; !COPY character-literal ; !SKIP [letter] ; !BEGIN [letter] ; !END ; !LINKAGE symbol ... ; !TRACE [trace-control] name ... ; !INTERFERENCE interference-control ; !REDUCIBLE ; !NOLIST ; !LIST ; !EJECT ; !BASE data-name integer-literal ; !ISBASE data-name integer-literal ; !DROP integer-literal ; !LEFTRIGHT ; !REARRANGE ; !INITIALIZE ; !ORDER ;
compool-list	[compool-file] name ,... ([compool-file])
compool-file	character-literal
trace-control	(boolean-formula)
interference-control	data-name : data-name ,...
data-name	item-name table-name block-name
symbol	letter ...

Notes:

1. A !COMPOOL directive can be given only immediately after a START or immediately following another !COMPOOL directive.

2. The names given in the !COMPOOL directive must be declared in the compool module designated by compool-file.

3. A name in a !COMPOOL directive cannot be the name of a component of a type-declaration, nor can it be the name of a formal parameter.

4. A !LINKAGE directive can only occur in a subroutine-declaration or subroutine-definition between the heading and the declarations of the formal parameters.

5. If a subroutine with a !LINKAGE directive is declared and defined, the !LINKAGE directive must appear in every declaration of the subroutine as well as in the definition.

6. All names in a !TRACE directive, including names used in the trace-control, except for statement names, must have been declared prior to their use in the !TRACE directive.

7. A !TRACE directive can only occur within a statement.

8. An !INTERFERENCE directive can occur only within a declaration.

9. A !REDUCIBLE directive can be placed only immediately following the semicolon of the subroutine-heading for a function.

10. If a function designated as reducible is both defined and declared, the !REDUCIBLE directive must appear in all the declarations as well as in the definition.

11. The !INITIALIZE directive can appear only in declarations, but not in an entry description or in a block-body or in a subroutine-declaration.

12. A block affected by an !ORDER directive cannot contain an overlay declaration.

13. The !ORDER directive must be given first in an entry-description or block-body.

program-body	{ simple-body compound-body }
simple-body	statement
compound-body	BEGIN [declaration ...] statement ... [subroutine-definition ...] [dir ...] [label ...] END
declaration	[dir ...] { simple-declaration compound-declaration }
compound-declaration	BEGIN declaration ... END
simple-declaration	{ data-declaration type-declaration subroutine-declaration inline-declaration statement-name-declaration external-declaration define-declaration overlay-declaration null-declaration }
data-declaration	{ item-declaration table-declaration block-declaration constant-declaration }
null-declaration	{ ; | BEGIN END }

item-declaration	ITEM item-name [STATIC] $\left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\}$ [item-preset] ;
type-description	$\left\{ \begin{array}{l} \text{integer-type-description} \\ \text{floating-type-description} \\ \text{fixed-type-description} \\ \text{bit-type-description} \\ \text{char-type-description} \\ \text{status-type-description} \\ \text{pointer-type-description} \end{array} \right\}$
integer-type-description	$\left\{ \begin{array}{c} S \\ U \end{array} \right\}$ [, $\left\{ \begin{array}{c} R \\ T \end{array} \right\}$] [integer-size]
floating-type-description	F [, $\left\{ \begin{array}{c} R \\ T \end{array} \right\}$] [precision]
fixed-type-description	A [, $\left\{ \begin{array}{c} R \\ T \end{array} \right\}$] scale [, fraction]
$\left. \begin{array}{l} \text{integer-size} \\ \text{precision} \\ \text{scale} \\ \text{fraction} \end{array} \right\}$	integer-ctf
item-name	name

Notes:

1. R indicates rounding. T indicates truncation.

2. A compile-time-formula is abbreviated in this syntax to ctf. Thus an integer-ctf is a compile-time-formula of type integer.

3. Only items with static allocation permanence can have a preset.

4. Integer-size must be greater than zero and less than or equal to MAXINTSIZE. If integer-size is omitted, BITSINWORD - 1 is assumed.

5. Precision must be greater than zero and less than or equal to MAXFLOATPRECISION. If precision is omitted, FLOATPRECISION is assumed.

6. The sum of scale and fraction must be greater than zero and less than or equal to MAXFIXEDPRECISION.

7. The value of scale must must lie in the range -127 through +127.

bit-type-description	B [bit-size]
char-type-description	C [char-size]
status-type-description	STATUS [status-size] (status-list)
status-list	{ default-list [default-list] specified-list }
default-list	status-const ,...
specified-list	status-group ,...
status-group	status-index { status-constant },...
status-constant	V ({ name letter reserved-word })
pointer-type-description	P [type-name]
bit-size char-size status-size status-index	integer-ctf
type-name	{ item-type-name table-type-name block-type-name }

Notes:

1. Bit-size must be greater than or equal to 1 and less than or equal to MAXBITS. If bit-size is omitted, 1 is assumed.

2. Char-size must be greater than or equal to 1 and less than or equal to MAXBYTES. If char-size is omitted, 1 is assumed.

3. The status-constants must be unique within a status-list.

4. All status-constants in a status-list must have a unique spelling.

5. In a default-list, the status constants are assigned representations starting with Ø and continuing to n-1, where n is the number of the status-constants in the default-list.

6. If status-size is omitted, a default status size that is the minimum necessary to represent the largest status constant is used to represent the value of all status-constants in a status-list.

<table>
<tr><td>table-declaration</td><td>TABLE table-name [table-attributes]
table-body</td></tr>
<tr><td>table-body</td><td>{ ; entry-description
table-type-name [table-preset] ;
unnamed-entry [table-preset] ; }</td></tr>
<tr><td>unnamed-entry</td><td>{ item-type-name
type-description } { packing-spec
table-kind }</td></tr>
<tr><td>table-attributes</td><td>[STATIC] [(dimension ,...)]
[structure-spec] [{ packing-spec
table-kind }]
[table-preset]</td></tr>
<tr><td>dimension</td><td>{ [lower-bound :] upper-bound
* }</td></tr>
<tr><td>lower-bound
upper-bound</td><td>{ integer-ctf
status-ctf }</td></tr>
<tr><td>structure-spec</td><td>{ PARALLEL | T [bits-per-entry] }</td></tr>
<tr><td>bits-per-entry</td><td>integer-ctf</td></tr>
<tr><td>packing-spec</td><td>{ N | M | D }</td></tr>
<tr><td>table-kind</td><td>{ W entry-size | V }</td></tr>
<tr><td>entry-size</td><td>integer-ctf</td></tr>
<tr><td>table-name</td><td>name</td></tr>
</table>

Notes:

1. The maximum number of dimensions is 7.

2. The * dimension can be used only with a table that is a formal parameter. If any dimension of such a table is *, then all dimensions must be *.

3. The lower-bound and upper-bound must both be integer formulas or both be status formulas.

4. If a lower-bound is not given, a lower-bound of Ø is assumed for an integer dimension and a lower-bound that is the first status-constant in the status list for a status dimension is assumed for a status dimension.

5. A packing-spec of N indicates no packing, M indicates medium packing, and D indicates dense packing.

6. A table-kind of W indicates a fixed-length-entry table and a table-kind of V indicates a variable-length-entry table.

7. If a table is declared in terms of a structure-spec or packing-spec, table-kind cannot be given.

8. If a table is declared in terms of a type-name, the preset is given following the type-name, not in the table-heading.

9. If T structure is specified for a table with a W table-kind, entry-size must not be given. The compiler uses the bits-per-entry either given or assumed as the size of the entry.

entry-description	{ simple-entry-description compound-entry-description }
simple-entry-description	{ table-item-declaration dir null-declaration }
table-item-declaration	ITEM item-name { item-type-name type-description } [{ packing-spec position }] [table-preset] ;
compound-entry-description	BEGIN simple-entry-description ... END
position	POS (starting-bit , starting-word)
starting-bit	{ integer-ctf | * }
starting-word	integer-ctf

Notes:

1. If the table-declaration contains a table-kind, position must be given for every item.
2. If the table-declaration does not contain a table-kind, packing-spec can be given.
3. If packing-spec is not given for such a table, each item begins in a new word.

block-declaration	BLOCK block-name [allocation-spec] { ; block-body block-type-name [block-preset] ; }
block-body	{ simple-block-body compound-block-body }
simple-block-body	{ data-declaration null-declaration }
compound-block-body	BEGIN { data-declaration overlay-declaration dir null-declaration } ... END
block-name	name

Notes:

1. No allocation order is implied by the order of the declarations within the block, unless an !ORDER directive is given within a compound-block-body.

2. The declaration of a constant can be given only in a block that has static allocation permanence.

3. A data-declaration within a block must not have an allocation-spec.

constant-declaration	{ constant-item-declaration / constant-table-declaration }
constant-item-declaration	CONSTANT ITEM constant-item-name { type-description / item-type-name } item-preset ;
constant-table-declaration	CONSTANT TABLE constant-table-name [(dimension-list)] [structure-spec] [{ packing-spec / table-kind }] [table-preset] table-body
constant-item-name } constant-table-name }	name

Notes:

1. Some of the items of a constant table must be set by a table-preset. That preset can be given in the table-attributes or as part of the table options.

2. The allocation permanence of all constant declarations, even those within subroutine definitions, is static.

3. The value of a constant item, except a pointer, can be used in a compile-time-formula. The value of a constant table or an item from a constant table cannot be used in a compile-time-formula.

item-preset	= item-preset-value
item-preset-value	{ ctf LOC (loc-arg) }
table-preset	= table-preset-value
table-preset-value	{ [positioner] preset-option } ,...
positioner	POS (index ,...) :
preset-option	{ item-preset-value repetitions (preset-option ,...) }
repetitions	integer-ctf
block-preset	= block-preset-value
block-preset-value	{ preset-option (table-preset-value) (block-preset-value) }

Notes:

1. An item must not be initialized more than once.

2. The type of the initial value must be compatible with the type of object being initialized.

3. If positioner is used, the indexes must correspond in type and number to the dimensions of the table.

4. Repetitions must be non-negative.

type-declaration	TYPE { item-type-declaration table-type-declaration block-type-declaration }
item-type-declaration	item-type-name { type-description item-type-name } ;
table-type declaration	table-type-name TABLE [(dimension-list)] { table-type-description table-type-name ; }
table-type-description	[structure-spec] [LIKE table-type-name] [{ packing-spec table-kind }] { ; entry-description unnamed-entry ; }
block-type-declaration	block-type-name BLOCK block-body
item-type-name table-type-name block-type-name }	name

Notes:

1. STATIC or presets cannot be given in a type-declaration.

2. A table can have only one dimension-list. The dimension-list can be given either in the table-declaration or in the type-declaration. Further, if a table-type-declaration contains a dimension-list, then it must not contain a table-type-name, either directly or in a like-option, that has a dimension-list.

3. The table-type-name in the like-option must agree in kind and structure with the table-type-declaration in which it is used.

<table>
<tr><td>subroutine-declaration</td><td>PROC sub-name [sub-attributes] ;
declaration</td></tr>
<tr><td>subroutine-definition</td><td>[dir ...]
PROC sub-name [sub-attributes] ;
[dir ..] subroutine-body</td></tr>
<tr><td>sub-attributes</td><td>[use-attribute] [(formal-list)]
[type-description]</td></tr>
<tr><td>sub-name</td><td>{ procedure-name | function-name }</td></tr>
<tr><td>use-attribute</td><td>{ REC | RENT }</td></tr>
<tr><td>formal-list</td><td>{ input-formals
[input-formals] : output-formals }</td></tr>
<tr><td>input-formals</td><td>{ data-name
statement-name
subroutine-name } ,...</td></tr>
<tr><td>output-formals</td><td>data-name ,...</td></tr>
<tr><td>subroutine-body</td><td>{ simple-body
compound-body }</td></tr>
</table>

inline-declaration	INLINE sub-name ,... ;
statement-name- declaration	LABEL statement-name ,... ;
statement-name	name

Notes:

1. If sub-attributes contain a type-description, then the subroutine being declared or defined is a function. Otherwise, it is a procedure.

2. A declaration must be given for all parameters in formal-list. Such declarations must not contain allocation-specs or presets and must not be external, constant, or type declarations.

3. The actual parameters in a subroutine call must match in number and kind (input or output) the formal parameters in the subroutine declaration or definition. Further, the actual and formal parameters must be compatible in type.

4. Item parameters that are input-formals are bound by value. Item parameters that are output-formals are bound by value-result. Parameters that are tables or blocks are bound by reference.

5. A subroutine must not be invoked recursively unless it is declared with the REC attribute. It must not be invoked reentrantly unless it is declared with the REC or RENT attribute.

6. A subroutine-body must contain at least one non-null statement.

7. Inline subroutines may themselves contain (possible inline) subroutine-calls, but not nested subroutine-definitions.

8. Names of subroutines whose definitions appear in other modules

external- declaration	{ def-specification ref-specification }
def-specification	DEF { simple-def | compound-def }
compound-def	BEGIN simple-def ... END
simple-def	[dir ...] { data-declaration statement-name-declaration null-declaration def-block-instantiation }
def-block- instantiation	BLOCK INSTANCE block-name ;
ref-specification	REF { simple-ref | compound-ref }
compound-ref	BEGIN simple-ref ... END
simple-ref	[dir ...] { data-declaration subroutine-declaration null-declaration }

Notes:

1. A data declaration in a def-specification and a correspon-ding declaration in a ref-specification must agree in name, type, and all attributes. However, the compiler checks this agreement only if a connection is established between the modules via a compool directive.

2. External data must have static allocation permanence.

3. The data-declaration in a def-specification or ref-specification cannot be a constant declaration.

4. In a ref-specification, presets are not allowed in the declaration of items or tables, and are permitted in the declaration of a block only if there is a corresponding DEF BLOCK INSTANCE.

5. For each subroutine-declaration in a ref-specification, a corresponding subroutine-definition, preceded by DEF, must exist in some procedure module.

overlay-declaration	OVERLAY [POS (overlay-address) :] overlay-string :... ;
overlay-address	number
overlay-string	{ spacer data-name (overlay-string :...) } ,...
spacer	W integer-ctf
define-declaration	DEFINE define-name [(define-formal-list)] [list-option] define-string ;
define-formal-list	define-formal ,...
define-formal	letter
list-option	{ LISTEXP | LISTINV | LISTBOTH }
define-string	" character ... "
define-call	define-name [(define-actual-list)]
define-actual-list	define-actual ,...
define-actual	{ [character ...] " [character ...] " }
define-name	name

Notes:

1. An overlay-declaration can only name data that is declared without a REF declaration in the same scope in which the overlay-declaration appears.

2. Declarations for all data names must precede the overlay-declaration.

3. An overlay-declaration within a block-declaration must not reference data names declared outside the block or within nested blocks and it must not contain an overlay-address.

4. An overlay-declaration outside a block-declaration must not reference data names declared within the block.

5. Define-actuals that are omitted are replaced by a null string. If the number of define-formals exceeds the number of define-actuals, a null string is substituted for each missing define-actual.

6. A quotation mark within a define-actual enclosed in quotation marks must be doubled.

7. If a define-declaration has a define-formal-list, then a define-call on the define-name must include the parentheses that enclose the define-actual-list, even though the list may be null.

8. A define-call cannot be juxtaposed with surrounding symbols to create, after substitution, a new symbol.

9. A define-call must not be used as the name being declared in a declaration or as a formal parameter within a subroutine heading.

statement	[label ...] [dir ...] { simple-statement compound-statement }
compound-statement	BEGIN [dir ...] statement ... [label ...] END
label	statement-name :
simple-statement	{ assignment-statement if-statement case-statement loop-statement exit-statement goto-statement procedure-call-statement return-statement abort-statement stop-statement null-statement }
null-statement	{ ; | BEGIN [label ...] END }

assignment-statement	variable-list = formula ;
variable-list	variable ,...
if-statement	IF test ; true-alternative [ELSE false-alternative]
test	boolean-formula
true-alternative false-alternative	statement
case-statement	CASE case-selector ; [dir ...] BEGIN [[dir ...] default-option] { [dir ...] case-option } ... [label ...] END
default-option	(DEFAULT) : statement [FALLTHRU]
case-option	(case-index ,...) : statement [FALLTHRU]
case-selector	formula
case-index	{ ctf | lower-bound : upper-bound }

Notes:

1. In an assignment-statement, the formula is evaluated and then the variables are evaluated and assigned the value of the formula, starting with the leftmost variable and proceeding from left to right to the rightmost variable before the equals sign.

2. The types of the variables in the list must be the same and the type of the formula must be compatible with this type.

3. The type of each case-index must be compatible with the type of the case-selector. The valid types for case-selector and case indexes are integer, bit, character, and status.

4. The values specified by the case-indexes must not overlap.

5. If a default-option is not given, the value of the case-selector must be represented by a case-index.

loop-statement	{ for-loop | while-loop }
for-loop	for-clause ; statement
while-loop	while-phrase ; statement
for-clause	FOR loop-control : initial-value [continuation]
continuation	{ while-phrase [by-or-then-phrase] by-cr-then-phrase [while-phrase] }
by-or-then-phrase	{ BY increment THEN next-value }
while-phrase	WHILE condition
loop-control	{ item-name | letter }
initial-value increment next-value	formula
condition	boolean-formula
exit-statement	EXIT ;
goto-statement	GOTO statement-name ;

Notes:

1. The while-phrase is performed before each execution of the statement within the loop and the by-or-then-phrase after the execution of that statement.

2. If loop-control is a letter, it must not be used in a context in which its value can be changed.

4. The type of initial-value, increment, and next-value must be compatible with the type of loop-control. The type of increment must be such that its value can be added to initial-value

5. If loop-control is a letter, initial-value must not be an ambiguous status-constant. A single letter loop-control is implicitly declared within the loop-statement. Its value is not known outside the loop-statement.

procedure-call	procedure-name [(actual-list)] [abort-phrase] ;
actual-list	{ input-actuals [input-actuals] : output-actuals }
input-actuals	{ formula statement-name subroutine-name block-reference } ,...
output-actuals	variable ,...
abort-phrase	ABORT statement-name
procedure-name	{ user-defined-procedure-name machine-specific-procedure-name }
return-statement	RETURN ;
abort-statement	ABORT ;
stop-statement	STOP [integer-formula] ;
machine-specific-procedure-name	name

Notes:

1. An abort-phrase must not be given in a procedure-call for a machine-specific procedure.

2. Actual parameters in the procedure-call must match the formal parameters of the called procedure in number, kind, and parameter list position.

3. The statement-name in an abort-phrase or input-actual must be known in the scope that contains the procedure-call-statement.

<table>
<tr><td>formula</td><td>{ operator operand
operand operator operand
(formula)
relational-expression }</td></tr>
<tr><td>operator</td><td>{ + | - | * | / | ** | MOD
AND | OR | NOT | XOR | EQV }</td></tr>
<tr><td>operand</td><td>{ literal
variable
constant
function-call
formula
conversion (formula)
implementation-parameter }</td></tr>
</table>

Notes:

1. The types of the operands in a formula must agree. The type of the formula is determined by the types of the operands.

2. NOT is a prefix operator. The operators + and - can be used as either infix or prefix operators. All other operators are infix operators.

3. The precedence of the operators is defined as follows:

Precedence	Operators
5	**
4	* / MOD
3	+ -
2	= <= >= <> < >
1	NOT AND OR EQV XOR

   If the operators in a logical formula are not the same, parentheses must be included to indicate the order of evaluation.

4. A formula cannot have two adjacent operators.

5. The following table gives the permissable operators and the formula type for each type of operand.

Operand	Operators	Formula Type
Integer	+ - * / ** MOD	S n-1 - where n is the actual number of bits supplied by the implementation for a signed integer item with the size attribute of the larger of the operands (for exponentiation, the formula type is integer only if the right operand is a non-negative integer compile-time-formula)..
Floating	+ - * / **	F p - where p is the precision of the most precise operand.
Fixed	+ - * /	A s,f - where s and f are as follows depending on the operator: + or - s equals the scale of the operands and f is the maximum of the fraction of the operands. * If one operand is a integer, s is the scale of the fixed operand and f is the fraction of the fixed operand. If both operands are fixed, s is the sum of the scales and f is the sum of fractions of the operands. / If the denominator is an integer, s and f are the scale and fraction of the numerator. If the denominator is a fixed type value, the result is exact and must be explicitly converted to a specified scale and fraction.
Bit	AND OR NOT EQV XOR	B n - where n is the number of bits in the longest operand.

relational-expression	formula relational-op formula
relational-op	$\{ < \mid > \mid = \mid <= \mid >= \mid <> \}$

Notes:

1. The type of the formulas in a relational expresssion must match.

2. When both formulas are status-constants, at least one must be unambiguously associated with a single status type.

3. When the two formulas are status formulas, their types must be identical.

4. When the two formulas are pointer formulas, their types must be identical or one must be an untyped pointer.

5. When both formulas are fixed formulas, a type to which both are automatically convertible must exist.

literal	{ integer-literal floating-literal fixed-literal bit-literal boolean-literal character-literal pointer-literal }
integer-literal	number
floating-literal fixed-literal	{ number exponent [number] . [number] [exponent] }
exponent	E [+ | -] number
number	digit ...
bit-literal	bead-size B ' { bead ... } '
bead-size	{ 1 | 2 | 3 | 4 | 5 }
bead	{ digit | A | B | ... | V }
digit	{ Ø | 1 | ... | 9 }
boolean-literal	{ TRUE | FALSE }
character-literal	' { character ... } '
pointer-literal	NULL

Notes:

1. In a floating- or fixed-literal, a number must be given either before or after the decimal point, or both before and after.

2. An integer-literal denotes a decimal value. Its type is S n, where n is IMPLINTSIZE(MINSIZE(integer-literal)).

3. A floating- or fixed-literal denotes a decimal value. Its type is determined from the context in which it is used, as follows:

Use	Type
preset	type of the object being preset
assignment value	type of the target being assigned the value
operand	type of the other operand
actual parameter	type of the formal parameter
initial-value for loop-control	type of the loop-control

4. TRUE represent the bit value 1B'1'. FALSE represents 1B'Ø'.

5. A ' character within a character-literal is represented by two adjacent ' characters.

variable	{ data-reference pseudo-variable function-name }
data-reference	name [(subscript-list)] [dereference]
subscript-list	index ,...
dereference	@ { pointer-name (pointer-formula) }
pseudo-variable	{ BIT (variable, first, number) BYTE (variable, first, number) REP (variable) }
constant	{ constant-name [(subscript-list)] control-letter }
cfirst cnumber }	integer-ctf
constant-name	{ constant-item-name constant-table-name }
pointer-name	name
index	{ integer-formula status-formula }

Notes:

1. Name in a data-reference and constant-name must be either an item-name or a table-name.

2. A subscript-list must be used to reference an entry of an item in a dimensioned table. The subscript-list must contain the same number of indexes as the dimension-list of the table contained dimensions.

3. Each index must agree in type with its corresponding dimension and be within the range specified by that dimension.

4. The indexes in a reference to a table that is a formal parameter declared with * dimensions must be integer formulas, even if the dimensions of an actual parameter are status types. The value of each index must be in the range Ø to n-1, where n is the number of entries in that dimension.

5. A dereference must be used to reference an item or table declared using a type-declaration.

6. In a BIT pseudo-variable, the argument variable must have type bit. In a BYTE pseudo-variable, the argument variable must have type character.

7. A pseudo-variable or function-name cannot be used as the argument of a REP function.

function-call	function-name [(actual-list)]
function-name	{ user-defined-function-name built-in-function-name machine-specific-function-name }
conversion	{ (* type-description *) (* type-name *) type-name type-indicator REP }
type-indicator	{ S | U | F | B | C | P }
machine-specific-function-name	name

Notes:

1. User-defined-function-names are those names defined to be functions in subroutine declarations or definitions.

2. Machine-specific-function-names are given in the user's guide.

built-in-function	{ LOC (loc-argument) NEXT (next-arg , incr)) BIT (bit-formula, first, number) BYTE (char-formula , first , number) SHIFTL (formula , count) SHIFTR (formula , count) ABS (formula) SGN (numeric-formula) BITSIZE (size-arg) BYTESIZE (size-arg) WORDSIZE (size-arg) LBOUND (table-name , dim-number) UBOUND (table-name , dim-number) NWDSEN (nwdsen-arg) FIRST (stat-arg) LAST (stat-arg) }
loc-arg	{ data-reference subroutine-name statement-name }
next-arg	{ pointer-formula | status-formula }
size-arg	{ formula | block-name | type-name }
nwdsen-arg	{ table-name | table-type-name }
stat-arg	{ status-formula | status-type-name }
incr first number count }	integer-formula

Notes:

1. The LOC of a subroutine whose name appears in an inline-declaration, or of a statement-name whose definition appears in such a sub-routine is implementation defined.

2. First and number must not designate a substring beyond the bounds of the bit or character formula.

3. Next-arg cannot be an ambiguous status-constant or the pointer literal NULL.

4. The type of the status-formula must be a status type with default representation.

5. When the next-arg is a status formula, the increment must not cause the NEXT function to return a value out of range of the next-arg.

6. The value of the pointer formula and the value of the pointer result must be in the implementation-defined set of valid values for pointers of its type.

name	{ letter | $ } name-char ...
name-char	{ letter | digit | $ | ' }
letter	{ A | B | C | ... | Z }
digit	{ Ø | 1 | 2 | ... | 9 }
character	{ letter | digit | mark | other-char }
mark	{ + | - | * | / | > | < | = @ | . | : | , | ; | (|) ' | " | % | ! | $ | blank }
reserved-word	{ ABORT | ABS | AND | BEGIN | BIT BITSIZE | BLOCK | BY | BYTE BYTESIZE | CASE | COMPOOL CONSTANT | DEF | DEFAULT | DEFINE ELSE | END | EQV | EXIT | FALLTHRU FALSE | FIRST | FOR | GOTO | IF INLINE | INSTANCE | ITEM | LABEL LAST | LBOUND | LIKE | LOC | MOD NEXT | NOT | NULL | NWDSEN | OR OVERLAY | PARALLEL | POS | PROC PROGRAM | REC | REF | RENT | REP RETURN | SGN | SHIFTL | SHIFTR START | STATIC | STATUS | STOP TABLE | TERM | THEN | TRUE | TYPE UBOUND | WHILE | WORDSIZE | XOR }
{ ctf integer-ctf integer-formula floating-formula status-formula pointer-formula }	formula

Notes:

1. Only the first 31 characters of a JOVIAL J73 name are used to determine uniqueness. Additional characters are ignored.

2. The other-chars are the remaining implementation-dependent characters accepted in character literals or comments. See the user's guide for a list of these characters and their collating sequence.

3. The following alternate characters are provided for implementations that do not have the given characters:

Standard Character	Alternate
@	↓ or ?
'	→ or _
"	≠
!	∨
%	[illegible]
:	%

Syntax Index

Appendix B

IMPLEMENTATION PARAMETERS

The way in which the memory of a machine is partitioned into addressable units is a fundamental part of the machine's architecture. JOVIAL (J73) assumes the following partitions:

Bit — The smallest unit of storage. It can contain one of two values, Ø or 1.

Byte — A group of bits that can hold 1 character of information.

Word — A group of one or more consecutive bits that serves as the unit of allocation of data storage.

Address Unit - The machine dependent unit used to identify a location or address in memory.

The number of bits per byte, per word, and per address varies from implementation to implementation. These quantities affect the representation and behavior of data in a high level language.

JOVIAL (J73) supplies implementation dependent parameters that allow these quantities to be referenced symbolically. The values of these constants must be specified in the user's guide for any implementation of JOVIAL (J73).

The following list gives the implementation parameters and a short description of the meaning of each. First the implementation parameters of type integer are given, then those of type float, then those of type fixed.

B.1 INTEGER IMPLEMENTATION PARAMETERS

The implementation parameters of type integer have the same size as an integer literal with the same value.

Paramter	Meaning
BITSINBYTE	Number of bits in a byte.
BITSINWORD	Number of bits in a word.
LOCSINWORD	Number of locations (address units) in a word.
BYTESINWORD	Number of complete bytes in a word.
BITSINPOINTER	Number of bits used for a pointer value.
FLOATPRECISION	Number of bits supplied to hold the value of the mantissa of a floating item declared with default precision.
FIXEDPRECISION	Number of bits, not including sign bit, supplied to hold the value of a fixed item declared with a default fraction.
FLOATRADIX	Base of the floating point representation, given as an integer.

IMPLFLOATPRECISION (precision) Number of bits, not including sign bit, in the mantissa for a floating point value with the given precision.

IMPLFIXEDPRECISION (scale, fraction) Number of bits, not including sign bit, used to represent an unpacked fixed item with the given scale and fraction. This value also determines the accuracy of fixed formula results.

IMPLINTSIZE (integer-size) Number of bits, not including sign bit, used to represent an unpacked U or S integer item with the given integer-size.

MAXFLOATPRECISION	Maximum precision that can be given for a floating item.
MAXFIXEDPRECISION	Maximum value for the sum of the scale and fraction of a fixed item.
MAXINTSIZE	Maximum size, not including sign bit, for signed and unsigned integers.
MAXBYTES	Maximum value for a character string item size. MAXBYTES must not exceed MAXBITS/BITSINBYTE.
MAXBITS	Maximum value for a bit string size. The maximum value of words per entry in a table is MAXBITS/BITSINWORD. The maximum BITSIZE of a table is MAXBITS.
MAXINT (integer-size)	Maximum integer value representable in integer-size + 1 bits, including sign bit.
MININT (integer-size)	Minimum signed integer value representable in integer-size + 1 bits, including sign bit, using the implementation's method of representing negative numbers.
MAXSTOP	Maximum value that can be given for an integer formula in a stop statement.
MINSTOP	Minimum value that can be given for an integer formula in a stop statement.
MAXSIGDIGITS	Maximum number of significant digits processed for a fixed or floating point literal.
MINSIZE (integer-compile-time-formula)	Minimum value of integer-size such that the value of the integer-compile-time-formula is less than or equal to MAXINT(integer-size) and greater than or equal to MININT(integer-size).

MINFRACTION (floating-compile-time-formula) Minimum value of n such that 2**(-n) is greater than the value of the floating-compile-time-formula.

MINSCALE (floating-compile-time-formula) Minimum value of n such that 2**n is greater than the value of the floating-compile-time-formula.

MINRELPRECISION (floating-compile-time-formula) Minimum value of precision such that FLOATRELPRECISION(precision) is less than or equal to the value of the floating-compile-time-formula.

B.2 FLOATING IMPLEMENTATION PARAMETERS

The implementation parameters of type float have the precision as an argument.

Parameter	Meaning Parameter
MAXFLOAT (precision)	Maximum floating point value that can be represented in the number of mantissa bits specified by precision, not including sign bit.
MINFLOAT (precision)	Minimum floating point value that can be represented in the number of mantissa bits specified by precision, not including sign bit, using the implementation's method of representing negative numbers.
FLOATRELPRECISION (precision)	Smallest positive floating point value that can be represented in the number of mantissa bits specified by precision, not including the sign bit, such that: 1.0 - FLOATRELPRECISION(precision) is less than 1.0 and 1.0 is less than 1.0 + FLOATRELPRECISION(precision).

FLOATUNDERFLOW (precision)   Smallest positive floating point value that can be represented in the number of mantissa bits specified by precision, not including sign bit, such that both FLOATUNDERFLOW(precision) and FLOATUNDERFLOW(-precision) are representable as floating point values.

B.3 FIXED IMPLEMENTATION PARAMETERS

The implementation parameters of type fixed have the scale and fraction specified as arguments.

Parameter	Meaning
MAXFIXED (scale, fraction)	Maximum fixed value that can be represented in scale+fraction+1 bits, including sign bit.
MINFIXED (scale, fraction)	Minimum fixed value that can be represented in scale+fraction+1 bits, including sign bit, using the implementation's method of representing negative values.

INDEX

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.