CYBIL

User's Guide

NOTICE:

This document is proprietary to Control Data. It is a restricted document for internal use only, and information contained herein is not to be disseminated outside the company.

DISCLAIMER:

This document is an internal working paper only. It is unapproved and subject to change, and does not necessarily represent any official intent on the part of CDC.

CONTENTS

CONTENTS

CONTENTS

PREFACE

This document explains the Control Data CYBIL programming language in a tutorial manner. The approach is to introduce simple concepts first, and then build upon these simple concepts and introduce more complex concepts in later sections. Each section introduces new material and it is assumed that the reader understands the material presented in the preceding sections.

An assumption is made that the reader is familiar with programming in general, and has some programming experience with at least one high-level language (e.g., FORTRAN, COBOL, ALGOL, etc.). It is not assumed that the reader is familiar with CYBIL or has attended a CYBIL programming course.

This document does not take the place of a reference manual nor does it describe features of the language that depend wholly on a particular implementation, except to mention their existence and treat them separately in an appendix. To make practical use of CYBIL the reader must have available additional reference material, including the CYBIL reference manual for the operating system to be used, and a description of the operating system's input/output capabilities and how they are accessed from CYBIL programs. Specifically, for NOS/VE and CYBER 180 development, the reader should have available (at minimum) the following documents, available through the Document Control System (DCS).

Document Name	DCS ID
Language Specification for CDC CYBIL	ARH2298
CYBIL Implementation Dependent Handbook	ARH3078
External Reference Specification for CYBIL I/O	ARH2739
External Reference Specification for Simulated NOS/VE I/O	ARH3125

CONVENTIONS

To distinguish CYBIL language elements and identifiers used in the examples from the English language text of this guide, several conventions have been adopted. Specifically,

- o Examples are presented as figures wherever possible, spatially distinct from surrounding text.

- o Within text, reserved words and identifiers are printed in full uppercase or enclosed in quotation marks.

1.0 INTRODUCTION

To be supplied.

2.0 LANGUAGE STRUCTURE

2.1 ALPHABET

The complete CYBIL alphabet consists of characters with graphic representations taken from the ASCII character set (see appendix A for the complete list). CYBIL uses a subset of the alphabet for specific purposes, such as defining an identifier.

2.2 CONSTANTS

Many types of constants are predefined to the compiler and indicate the appropriate value. For example, integer constants are known to the compiler. To represent the constant twenty-five, the programmer writes 25. Negative integer constants may also be used (for example, -33). Some constants are programmer defined (for example, string constants and ordinal constants; see section 3).

Boolean constants (TRUE and FALSE) may be used and have a predefined meaning to the compiler (refer to section 3, Boolean Type).

Character constants are defined in terms of the ASCII character set. A character constant is indicated by placing the character within apostrophes (for example, 'B' refers to the character constant uppercase B). Refer to section 3, Character Type.

A string constant consists of one or more adjacent characters enclosed in apostrophes (for example, 'ABCD1234' is a string constant consisting of eight characters). String constants are always programmer defined. Refer to section 8 for examples illustrating the use of string constants.

The pointer constant NIL indicates an unassigned pointer. NIL can be assigned to a pointer variable of any type. Refer to section 11 for a discussion of pointers.

Ordinal constants are identifiers defined by the programmer. Refer to section 3, Ordinal Type.

2.3 IDENTIFIERS

CYBIL uses reserved words (described in appendix D), special marks (for example, + - /), digraphs or pairs of special marks (for example, := <= <>), and programmer defined identifiers. Reserved words, special marks, and digraphs are predefined to the CYBIL compiler and are not considered to be identifiers.

The programmer may define identifiers as needed for module names, procedure names, type names, variable names, constant names, and label names (all of which are defined in later sections of this guide). The programmer-defined identifier may not be the same as a reserved word. A quick glance at appendix D will familiarize the reader with the reserved words. Many simple programming errors involve conflicts with reserved words.

Programmer defined identifiers are limited in length to 31 characters. Uppercase and lowercase characters (for example, A and a) are treated as the same character when used in an identifier. For example, CYBIL treats NEWPROGRAMTEXT and NeWpRoGrAmTeXt as the same identifier.

The first character of an identifier must be alphabetic (that is, A thru Z or a thru z). Valid subsequent characters include A thru Z, a thru z, the digits 0 thru 9, and four additional characters: underline (_), number sign (#), dollar sign ($), and commercial at (@).

Examples of identifiers are shown in figure 2-1.

Valid	Invalid
wheat_production	3rd_Test
A@10	ONE+ONE
Syntax_Table	@SIGN
Z#_$@	_LARRY_
ABC	Field_3.7
PETE___	I/O
TEST_deck3#215	
NAMEPOINTER	
Field_A	

Figure 2-1. Identifiers

The valid identifiers in figure 2-1 need no explanation. The invalid identifiers are incorrect for the following reasons: 3rd_Test, first character not alphabetic; ONE+ONE, contains an illegal character (+); @SIGN, first character not alphabetic; _LARRY_, first character not alphabetic; Field_3.7, contains an illegal character (.); I/O, contains an illegal character(/).

2.4 USE OF BLANKS

Identifiers, reserved words, and digraphs cannot contain embedded blanks. Elsewhere, blanks may be used freely, except in string constants where a blank represents a character.

2.5 COMMENTS

A comment may be used anywhere that blanks may be used (except in string constants). A comment is printed in the source listing but does not alter the meaning of the source program (that is, comments are ignored by the compiler).

A comment is bounded by left and right braces: { and }. A comment may contain any character except a right brace (}). Comments that cross line boundaries must be restarted at each line as shown in figure 2-2.

```
{THIS COMMENT
{CROSSES LINE
{BOUNDARIES.}
```

Figure 2-2. Comment Example

If you forget to terminate a comment with a right brace (}), CYBIL terminates the comment when it encounters the end of the line.

2.6 STATEMENT SEPARATOR (SEMICOLON)

The semicolon (;) is used to separate one statement from another. Extra semicolons may be used and indicate one or more empty statements. Since CYBIL ignores empty statements, extra semicolons have no effect on program execution (and do not cause compilation errors).

2.7 MODULE STRUCTURE

Every CYBIL program must be bounded by the reserved words MODULE and MODEND. The module's name (an identifier) must be specified after MODULE and can be repeated after MODEND.

```
MODULE sample;
    .
    .
{CYBIL source
{text
    .
    .
MODEND sample
```

Figure 2-3. Module Declaration

The acceptable structure illustrated in figure 2-3 shows a CYBIL module declaration which consists of a CYBIL source program enclosed by MODULE and MODEND. The structure in figure 2-3 also shows the use of an identifier (SAMPLE) to provide a name for the module. It is good practice to provide meaningful names for modules to enhance the readability of the source text.

Additional implications of the MODULE and MODEND statements with respect to program structure and scope of identifiers are described in section 13.

2.8 COMPILATION_UNITS

The compilation unit consists of a module declaration and optional compile-time statements and comments. Module declaration is described more fully in section 13. Compile-time declarations and statements are discussed in section 14. A pictorial representation of a compilation unit is given in figure 2-4.

```
             +--------------------------------+       ----+
             :    .                           :           :
             :    .                           :           :
             :    {Compile-time statements    :           :
             :    {and comments               :           :
             :    .                           :           :
             :    .                           :           :
             : MODULE sample_prog;            :--+        :
          +--:    {Additional compile-time    :  :        >Compilation
          :  :    {statements and             :  :        :   Unit
  Source  :  :    {comments                   :  :        :
  Program<   :    {CYBIL source text}         :  >Module  :
          :  :    .                           :  :        :
          :  :    .                           :  :        :
          +--:    .                           :  :        :
             : MODEND sample_prog             :--+        :
             +--------------------------------+       ----+
```

Figure 2-4. Compilation Unit

A number of compilation units may exist, one after another, in the source input file. The compiler reads the source input file and produces one or more object modules on the object file. The number of object modules produced will be equal to the number of compilation units if there are no errors during the compilation process. Certain errors (within one compilation unit) may suppress the corresponding object module. In general, one object module is produced for each compilation unit. It should be clear that a compilation unit and a module are not identical. Each compilation unit contains one module declaration.

The compilation process, then, transforms compilation units on the source input file into object modules on the object file (see figure 2-5).

```
Source Input File                                    Object File
+---------------+                                    +----------+
| +-----------+ |                                    | +------+ |
| |Compilation| |                                    | |Object| |
| |   Unit    | |           +---------+              | |Module| |
| +-----------+ |           |         |              | +------+ |
|       .       |           | CYBIL   |              |    .     |
|       .       |--------->|Compiler|---------->|    .     |
|       .       |           |         |              |    .     |
| +-----------+ |           +---------+              | +------+ |
| |Compilation| |                                    | |Object| |
| |   Unit    | |                                    | |Module| |
| +-----------+ |                                    | +------+ |
+---------------+                                    +----------+
```

Figure 2-5. Compilation Process

2.9 BLOCK STRUCTURE

In CYBIL, the programmer can create and identify unique blocks (groups) of source statements. The purpose of blocks is primarily to provide a structure to a sequence of statements. For example, a given program performs input, computes a result, and prints the result. An obvious organization for this program consists of three blocks. To change the computational method used in this sample program, the programmer concentrates his efforts on the computational block. If the program were not block structured, the programmer must first find the source lines dealing with the computation. This may not be easy, since the computational instructions might not be grouped together.

In addition to simple grouping, CYBIL provides for the declaration of variables inside blocks. These variables are considered to be local to the block in which they are defined. The programmer can use this block structure to manage storage requirements. When execution of a program enters a block, memory (storage space) is set aside for the variables declared in the block. When the program finishes a block, the storage space reserved for the local variables is released.

One advantage of this approach is that storage space for variables exists only when it is needed. A disadvantage is that extra storage management code must be generated and later executed to accomplish this management function, thereby slowing both compilation and execution, as well as increasing storage requirements for the program itself.

A CYBIL program may be written with lots of block structure or little block structure. The choice is up to the programmer. Liberal use of block structure makes a source program easier to maintain or change, and for this reason, use of block structure is recommended wherever possible.

Block structure is introduced by the use of procedure declarations, described in section 9.

2.10 SCOPE OF IDENTIFIERS

The scope of an identifier is the domain of a CYBIL program over which all references to an identifier are associated with the same definition of that identifier. The scope of identifiers is affected by the use of blocks (procedure declarations). An identifier may be referenced only in the block in which it is declared and blocks contained within the defining block. There are a few methods for avoiding this rule (known as scope attributes); they are discussed in section 3.

```
             Flow of Program Execution
                        :
                        :
                        :
                        :       Block 1
      +-----------------:-----------------+
      :Identifier A     :                 :
      :                 v Block 2         :
      :        +------------------+       :
      :        :Identifier B      :       :
      :        :                  :       :
      :        :                  :       :
      :        +--------+---------+       :
      :                 :                 :
      :                 :                 :
      :                 :                 :
      :                 v Block 3         :
      :        +------------------+       :
      :        :Identifier C      :       :
      :        :                  :       :
      :        :                  :       :
      :        +--------+---------+       :
      :                 :                 :
      :                 :                 :
      +-----------------:-----------------+
                        :
                        v       Block 4
      +-----------------------------------+
      :Identifier D                       :
      :                                   :
      :                                   :
      +-----------------+-----------------+
                        :
                        :
                        v
```

Figure 2-6. Scope of Identifiers

Four blocks appear in figure 2-6. Each block contains an identifier (A, B, C, and D). These identifiers could be names of types, variables, constants, or procedures. The particular kind of identifier is not important to this discussion.

Identifiers can be local or global, depending upon their position in the module and the point of reference. The following discussion describes where a reference to a given identifier is valid.

The identifier A is defined in, or local to, block 1. Identifier A is global to blocks 2 and 3, which are contained within block 1 (the defining block). Source program statements may refer to (make valid references to) all variables local to their block and also to variables global to their block. So, for example, statements in block 1 can refer to identifier A, and so can statements in blocks 2 and 3. However, statements in block 4 cannot refer to identifier A.

Identifier C is local to block 3. Statements in block 3 can refer to identifier C because it is local to block 3, and to identifier A because it is global to block 3. However, statements in block 3 may not refer to identifiers B or D because these identifiers (B and D) are neither local nor global to block 3.

Table 2-1 indicates the scope of the identifiers illustrated in figure 2-6.

TABLE 2-1. SCOPE OF IDENTIFIERS

Identifier	Defined in and local to	Global to	Can be referenced by statements in
A	Block 1	Blocks 2 and 3	Blocks 1, 2, and 3
B	Block 2	none	Block 2 only
C	Block 3	none	Block 3 only
D	Block 4	none	Block 4 only

Figure 2-6 shows that block 1 is entered first. If A denoted a variable, CYBIL would find space for variable A. Next, block 2 is entered. If B denoted a variable, CYBIL would find storage space for variable B. Next, an exit from block 2 occurs. Upon exit from block 2, the space reserved for variable B is released. Note that the space for variable A is not released.

Upon entry to block 3, space is reserved for variable C (assuming C is a variable). Upon exit from block 3, the space reserved for variable C is released. Upon exit from block 1, space reserved for variable A is released. The process continues for variable D as block 4 is entered and exited.

Notice that each of the variables (A, B, C, and D) has a specific lifetime. Each comes into existence upon block entry and disappears upon the appropriate block exit.

For this reason, variables declared within blocks are called automatic variables. That is, CYBIL automatically establishes storage space for these variables upon block entry and releases it upon block exit.

2.11 REDEFINED IDENTIFIERS

When an identifier name is defined in more than one block, CYBIL uses well formed rules to determine which definition of the identifier applies to a particular reference. The following paragraphs describe what happens when a reference is made to an identifier for which both local and global definitions exist, as illustrated in figure 2-7.

```
  Flow of Program Execution
               :
               :
               :      Block 11
+--------------:--------------+
:Identifier X  :              :
:              v Block 12     :
:       +--------------+      :
:       :Identifier X  :      :
:       :              :      :
:       :              :      :
:       +------+-------+      :
:              :              :
:              :              :
:              :              :
:              v Block 13     :
:       +--------------+      :
:       :Identifier Y  :      :
:       :              :      :
:       :              :      :
:       +------+-------+      :
:              :              :
:              :              :
:              :              :
+--------------:--------------+
               :
               v      Block 14
+-----------------------------+
:Identifier X                 :
:                             :
:                             :
+--------------+--------------+
               :
               :
               v
```

Figure 2-7. Identifier Conflicts

In figure 2-7 identifier X is declared in three different blocks (11, 12, and 14). Assume that all identifiers (X's and Y) are variables. Tracing the flow of program execution shows that block 11 is entered first where storage space is obtained for the variable identifier X in block 11. Next, block 12 is entered and space is obtained for the variable X declared in block 12.

At this point, there are two variables identified X in existence. Statements within block 12 can refer only to the variable identifier X local to block 12. When an exit occurs from block 12, the block 12 local variable X is released, but the block 11 variable X remains. Statements in block 13 can reference variable X from block 11.

Finally, when the flow of program execution exits block 11, space for the local variable X in block 11 is released. Then, block 14 is entered and a new variable identifier X for block 14 is established.

While all this may seem a little confusing at first, there are some benefits derived from block structure. Because of the way CYBIL treats local variables and identifiers, blocks can share variables and identifiers when they are global to the blocks. Identifiers can be as global as necessary to provide the amount of sharing needed (for example, identifier A in block 1, figure 2-6). Conversely, by declaring a variable or identifier within an inner block (for example, identifier Y in block 13, figure 2-7) or within a separate block (for example, identifier X in block 14, figure 2-7), the programmer shields the space reserved for the variable identifier from being referenced from any other block. This protection can improve program reliability and ease the burden of making program corrections. When an identifier is shielded from other blocks, a programmer can make changes to the local (shielded) variable with full confidence that he is not destroying an essential variable referenced elsewhere in the program.

3.0 VARIABLES AND CONSTANTS

3.1 VARIABLE DECLARATION FORMAT

All variable declarations follow the general format shown in figure 3-1.

```
VAR identifier : [attributes] type := initialization
```

Figure 3-1. Variable Declaration Format

When the attributes and initialization (refer to figure 3-1) are omitted, the variable declaration format takes the simple form shown in figure 3-2.

```
VAR identifier : type
```

Figure 3-2. Simplified Variable Declaration

The reserved word VAR introduces the variable declaration. The programmer declared identifier (or name) used to refer to the variable is given next, followed by the type of the variable.

3.2 TYPE

In the variable declaration each variable identifier is associated with a type. The type of the variable determines the operations allowed with the variable. With this information the compiler performs many compile time checks, resulting in more comprehensive error checking. For example, the assignment of an integer constant to a variable is valid only if the variable is declared to be type integer (or a subrange of the integers). Any other assignment is illegal. This may seem trivial in the case of type integer; however, checking types at compile time can be extremely valuable when dealing with pointers or structured types.

3.3 SCALAR TYPES

A scalar type is one in which the values can be ordered (scaled). The scalar types are integer, character, boolean, ordinal, and subrange. Scalars have the property that given any specific value one can, in general, find the next value (successor) or the preceding value (predecessor) of the scalar. For example, given the integer eight, the predecessor is seven and the successor is nine. A scalar variable is a variable declared to have a scalar type.

3.3.1 INTEGER TYPE

The integer type (INTEGER) consists of positive and negative integers in the range allowed by the computer hardware. Appendix B discusses these limitations. The folowing statement declares a variable COUNT to be type INTEGER.

```
VAR count : integer;
```

Figure 3-3. Integer Variable Declaration

Notice (in figure 3-3) the use of the reserved word VAR to introduce the declaration. The identifier (COUNT) appears next, separated from the type (INTEGER) by a colon. The semicolon separates this declaration from subsequent declarations or statements.

To declare three variables (COUNT, ROOT, and XSQUARE) to be integer one could write three VAR statements with one declaration in each statement or one VAR statement with three declaration parts.

Three_Declarations	One_Declaration

`VAR count : integer;`
`VAR root : integer;`
`VAR xsquare : integer;`
	`VAR`
	`  count : integer,`
	`  root : integer,`
	`  xsquare : integer;`

Figure 3-4. Multiple Variable Declarations

In figure 3-4, the three-declaration approach requires three VAR reserved words and each VAR declaration is separated from the next declaration by a semicolon.

With the one-declaration approach, only one VAR statement appears. It occupies four lines but it is one declaration (starting with VAR and ending at the semicolon). The declarations for COUNT, ROOT, and XSQUARE are separated by the comma. This form of the variable declaration eliminates the need to write the reserved word VAR over and over again.

CYBIL allows an additional form of the variable declaration. When many variable identifiers are the same type, the identifiers may be written on the left of the colon separated from each other by commas (see figure 3-5).

```
VAR
  count, root, xsquare : integer;
```

Figure 3-5. Multiple Variable Declarations

When declared in a block, and in the absence of any attributes (see figure 3-1), the variable declarations in figures 3-3, 3-4, and 3-5 are automatic variables. The system allocates space for these variables when the block in which the variables are declared is entered at execution time. The storage space for these automatic variables disappears when an exit from the containing block occurs. These rules define the lifetime of the variables. The scope of the variable identifiers is the block containing the declarations.

3.3.2 CHARACTER TYPE

The character type (abbreviated CHAR) consists of a character set of 256 distinct characters; the first 128 are ASCII, the remainder are unassigned. Some characters have graphic representations (for example, the letters A through Z, a through z, the numbers 1 through 9); many do not. The ASCII character set and associated values are illustrated in appendix A.

When a variable is declared to be type CHAR it may assume (be assigned) any character value. A character value is represented by the character graphic enclosed in apostrophes (for example, 'A' to represent the character value A), or by the value of a conversion function described in section 5 (CHR). A value of type other than character may not be assigned to a character variable. This kind of error is detected during program compilation.

The statement in figure 3-6 declares a variable (MIDDLEINITIAL) to be type character.

```
VAR
  middleinitial : char;
```

Figure 3-6. Character Variable Declaration

The rules governing variable lifetime and identifier scope are the same as described for the integer variable.

3.3.3 BOOLEAN TYPE

The boolean type (BOOLEAN) provides a mechanism for representing logical values. The two boolean values are TRUE and FALSE. TRUE and FALSE are reserved identifiers and represent the values associated with the boolean type. A variable declared to be type boolean may be assigned one of two values (TRUE or FALSE). Any other assignment (such as the assignment of an integer value) to a boolean variable is an error detected during compilation. Variable identifiers INHIBIT_READ and NOT_BUSY could be declared to be boolean variables with the statement shown in figure 3-7.

```
VAR
  inhibit_read, not_busy : boolean;
```

Figure 3-7. Boolean Variable Declarations

Operations that produce boolean results are discussed further in other sections.

3.3.4 ORDINAL TYPE

An ordinal is an ordered (scalar) sequence of user-defined identifiers. These ordinal identifiers are then values which may be assigned to the ordinal variable.

The ordinal type is a convenient way of introducing meaning into a program. For example, assume that, in writing a program to drive a line printer, a variable (STATUS) is needed to contain the current status of the printer. For this particular printer there are four possible statuses: ready, not ready, parity error, and lost data. In a conventional programming language one might declare a variable STATUS to be type integer and assign a value (1 to 4) to represent the actual status.

Now consider program maintenance (sometime in the future). When a statement says assign the value 3 to the variable STATUS, the meaning may not be clear. Either program comments or auxiliary documentation are needed to specify that the value 3 represents the parity error status. CYBIL offers a better solution, shown in figure 3-8.

```
VAR
  status : (ready, not_ready, parity_error, lost_data);
```

Figure 3-8. Ordinal Variable Declaration

Figure 3-8 illustrates how an ordinal is declared. An ordinal is an ordered list of (user-defined) identifiers enclosed in parentheses. In this example, the variable STATUS is declared to be of ordinal type. The specific possible ordinal values (user defined constants) are written in parentheses. The term ordinal refers to the ordered list of user-defined identifiers; the term ordinal identifier (or ordinal constant) refers to a particular identifier used to define an ordinal.

Ordinal identifiers are declared in ascending order. Once declared, they cannot be used in any way other than to denote an element of the ordinal (within the scope of the ordinal declaration). Given an ordinal identifier, say PARITY_ERROR, it is possible to find the predecessor (NOT_READY) and the successor (LOST_DATA) just like any other scalar type. The major difference between ordinals and other scalar types is that the programmer defines the values constituting an ordinal by the order in which they are specified.

The first ordinal identifier (for example, READY in figure 3-8) may be thought of as having the value zero, the second identifier (NOT_READY in figure 3-8) has the value one, and so on. Section 5, Scalar Functions, discusses explicit methods of translating between an ordinal constant and its equivalent value.

3.3.5 SUBRANGE OF SCALAR TYPES

When declaring a variable, one does not always want the variable to be able to take on all possible values normally associated with the type. Sometimes it is desirable to define a subrange of a given (scalar) type.

For example, assume the task is to write a payroll program. All the employees have been assigned employee numbers and the payroll department guarantees that employee numbers are always integers that never exceed 5 digits. A variable identifier (EMPLOYEE_NO) declared to be type integer could be mistakenly assigned an illegal employee number (any integer value greater than 99999). A better declaration for the employee number is shown in figure 3-9.

```
VAR
  employee_no : 1 .. 99999;
```

Figure 3-9. Subrange of Integer

Instead of a type (INTEGER, in this case), the declaration in figure 3-9 specifies 1 .. 99999. This means that the variable EMPLOYEE_NO is limited in range to the values between 1 and 99999 inclusive. The compiler will generate object code to check that all assignments to variable EMPLOYEE_NO fall in this range. If an assignment is found (at execution time) to be outside this range, an execution time error will occur and the program will terminate. The execution time error checking may be optionally disabled (refer to section 14). If CYBIL detects an invalid assignment during compilation, it issues an appropriate diagnostic message.

The symbol .. , in figure 3-9, defines a subrange. A scalar value (not necessarily an integer) must appear on the left and right of the .. symbol. The left value must be less than or equal to the right value. The type of the subrange is assumed from the type of the left or right argument. Either left or right may be used since they must both be the same type. For example, in figure 3-9, EMPLOYEE_NO is said to be a subrange of the type integer.

Subranges of other types are also possible. Suppose that a variable were needed that could be assigned any uppercase letter (not any character). This could be accomplished as shown in figure 3-10.

```
VAR
  alphabetic : 'A' .. 'Z';
```

Figure 3-10. Subrange of Character

The .. (in figure 3-10) indicates that the type is to be a subrange. The 'A' and 'Z' indicate the lower and upper bounds of the subrange. This subrange is a subrange of characters. Note that the character A is specified by enclosing the character in apostrophes ('A').

A subrange of type boolean is possible but not too meaningful as shown in figure 3-11.

```
VAR
  data_flag : FALSE .. TRUE;
```

Figure 3-11. Subrange of Boolean

Since there are only two values associated with type boolean (FALSE and TRUE) the type of DATA_FLAG in figure 3-11 is really boolean. In the scalar ordering, FALSE is defined (by the CYBIL specification) to precede TRUE.

CYBIL also permits subranges of ordinals.

```
VAR
  hardware : (tacks, nails, spikes, bolts, nuts),
  hammer_stuff : tacks .. spikes;
```

Figure 3-12. Subrange of Ordinal

In figure 3-12, an ordinal variable, HARDWARE, is declared to contain TACKS, NAILS, SPIKES, BOLTS, or NUTS. The stuff that can be hit with a hammer (HAMMER_STUFF) is declared to be a subrange of the preceding ordinal (TACKS .. SPIKES).

In all subrange declarations the values included in the subrange can be determined (by the compiler or programmer) by finding the successor of each value starting with the first (TACKS in figure 3-12) and ending with the last (SPIKES).

3.4 INITIALIZING VARIABLES

Many programming situations require a variable to have a particular value when program execution begins. This is done by specifying an initial value for the variable when it is declared (refer to figure 3-1 for the general variable declaration format).

```
VAR
  voltage : (low, medium, high) := medium,
  next_char : char := 'B',
  initial_value : integer := -3721946,
  year : 1900 .. 2000 := 1978,
  first_pass : boolean := TRUE;
```

Figure 3-13. Initialization of Variables

After the variable identifier and type (in figure 3-13), an initial value is specified after the := symbol. The initial value must be the proper type for the variable declared, and must be in the proper range.

VOLTAGE is a variable (in figure 3-13) declared to be an ordinal. The initial value is the ordinal value MEDIUM. The other variables in figure 3-13 need no explanation.

3.5 ATTRIBUTES

Attributes are used to control the method of access, method of storage, and scope of programmer defined variables. Refer to figure 3-1 to review how attributes are specified.

3.5.1 ACCESS ATTRIBUTE (READ)

The access attribute (READ) is used to indicate read-only access for a variable. Assignments to read-only variables are not allowed. A read-only variable must be initialized as this is the only way to provide a value.

```
VAR
  loop_limit : [READ] integer := 25,
  end_char : [READ] char := '*';
```

Figure 3-14. Access Attribute

In the example above (figure 3-14), the variable LOOP_LIMIT is defined to be a read-only variable of type integer with the value 25. Subsequent program statements (within the scope of this variable identifier) may access (refer to) the identifier LOOP_LIMIT whose value is 25. No statement may make an assignment to (or alter) the value of this variable.

(Read-only variables are usually structured variables, described in section 8. Constant declarations, described later in this section, are usually more suitable than read-only scalar variables.)

3.5.2 STORAGE ATTRIBUTES (STATIC AND SECTION)

When a variable is given the static attribute, normal storage allocation and returning (associated with automatic variables) is not performed. In effect, this makes a variable nonautomatic. The storage for a static variable is obtained no later than the entry to the block containing the declaration.

Unlike automatic variables, the storage space for a static variable is not returned upon block exit. The storage space is static. The static attribute can be used to extend the lifetime of a variable. An example might be a counter whose value indicates how many times that the block in which it is declared is entered.

Only a static variable declaration may contain an initialization. If so, the initialization occurs only once, when the storage space is first made available.

The scope of the static variable, however, does follow normal scope rules. That is, the identifier may be accessed only in the block containing the declaration (and nested blocks). An example of the static attribute is shown in figure 3-15.

```
VAR
  block13_count : [STATIC] integer := 0,
  master_flag : [STATIC] boolean;
```

Figure 3-15. Static Attribute

3.5.3 SCOPE ATTRIBUTES (XDCL AND XREF)

Scope attributes extend the scope of an identifier defined in a variable declaration statement. Any variable given a scope attribute automatically has the storage attribute STATIC. The programmer need not specify the static attribute.

Normally, the scope of a variable is the block in which the identifier is declared. If the identifier is to be used in many blocks, it is made global enough to accomplish this objective. But the scope of the most global identifier (without a scope attribute) is still the module in which it is declared.

Scope attributes allow an identifier to be known (or shared) between modules. When the loader loads the modules, it links together the variables with scope attributes.

The XDCL attribute indicates that an identifier is declared in a module and may be referenced from any other module. Variables with the XDCL attribute may be initialized and assignments are allowed.

The XREF attribute indicates that an identifier is declared in some other module and is referenced from this module. Variables with the XREF attribute may not be initialized, but assignments are allowed.

When the loader loads modules, it resolves XDCL and XREF variables. XDCL variables are allocated space (as static variables) and XREF variables in other modules with the same identifier share the same storage space as the equivalent XDCL variable.

The loader issues an error message if an XREF variable has no XDCL counterpart.

```
 MODULE first;                       MODULE last;
   .                                    .
   .                                    .
   VAR                                  VAR
     count : [XDCL] integer := 0;         count : [XREF] integer;
   .                                    .
   .                                    .
 MODEND first;                       MODEND last;
```

Figure 3-16. Scope Attributes (XDCL and XREF)

In figure 3-16, two modules are compiled producing two separate object modules. The loader then loads both modules prior to program execution. The variable COUNT in module FIRST is type integer, XDCL, initialized to the value zero, and static by default.

When the loader encounters variable COUNT in module LAST, it assigns the same storage space as that used for COUNT in module FIRST. In this way, references to variable COUNT from either module will refer to the same storage location(s).

Variable identifiers used with XDCL and XREF attributes must conform to the loader requirements for identifiers.

3.6 CONSTANT DECLARATION FORMAT

All constant declarations follow the general format shown in figure 3-17.

```
CONST identifier = constant expression;
```

Figure 3-17. Constant Declaration Format

The constant declaration is introduced with the reserved word CONST. Note the use of the equal sign between the identifier and the constant expression.

The constant identifier follows the same scope rules that apply to other identifiers. The constant expression is evaluated at compile time and is associated with the constant identifier. The constant identifier may then be used (subject to scope rules) instead of the constant expression.

A constant cannot be altered by subsequent assignment in the program. It is, in effect, a read only constant.

In a program there may be many references to values known at compile time. Examples include the lengths of arrays and strings, the number of iterations to perform, and so on.

A good programming practice is to declare constants where appropriate, and then use the constant identifier throughout the program (instead of the constant value). This approach makes changing the constants in the program easy. All one needs to do is change one constant declaration at one place in the source text.

Some sample constant declarations are illustrated in figure 3-18.

```
CONST
  one = 1,
  unity = 1,
  number_of_elements = 3721,
  array_size = (number_of_elements + 9) / 10
  first_letter = 'A',
  flag_up = TRUE;
```

Figure 3-18. Constant Declarations

The type of the constant identifier is the same as the type of the constant expression. For example, constants ONE and UNITY are type integer, constant FIRST_LETTER is type character, and constant FLAG_UP is type boolean.

4.0 TYPE_DECLARATION

Section 3 discussed variable identifiers and the syntax for establishing a type for the variable identifiers. A scalar variable was defined to be one whose corresponding type was a scalar type (that is, integer, character, boolean, ordinal, or subrange).

CYBIL provides a mechanism to separate the definition of a type from the declaration of variables. In addition to the CYBIL predefined scalar types (integer, character, and boolean), the programmer can define a type and establish a type identifier. When variables are declared, their type must always be specified. But now, the variable may be a CYBIL predefined scalar type (for example, INTEGER, BOOLEAN, CHAR) or a programmer declared type. The general format of a type declaration is shown in figure 4-1.

```
TYPE identifier = type specification
```

Figure 4-1. Type Declaration Format

The reserved word TYPE introduces the type declaration statement. The identifier must conform to the rules governing length and valid characters for identifiers. The type specification is a programmer declared (valid CYBIL) type.

The type declaration is a compile time declaration. It does not occupy storage space during program execution. The type declaration does follow the rules of scope of identifiers. A type identifier may be referenced (at compile time) only in the block in which it is declared and blocks contained in the defining block.

Figure 4-2 illustrates a simple use of the type declaration.

```
TYPE
  tape_position = (beginning, middle, last);
    .
    .
VAR
  scratch_tape : tape_position := beginning,
  out_tape : tape_position := last;
    .
    .
```

Figure 4-2. Type Declaration Example

In the example in figure 4-2, a type identifier (TAPE_POSITION) is declared to denote an ordinal whose values are BEGINNING, MIDDLE, and LAST. The type identifier (TAPE_POSITION) means (or stands for) the ordinal wherever it is used within the block in which it is declared. The variable SCRATCH_TAPE has been declared to be type TAPE_POSITION and will be initialized to the ordinal value BEGINNING. The variable OUT_TAPE is also declared to be type TAPE_POSITION and is initialized to the ordinal value LAST.

In the variable declaration one could write the ordinal itself (the parenthesized list of user-defined identifiers) instead of a reference to the defined ordinal type (TAPE_POSITION). This approach would require that the ordinal be written twice, as shown in figure 4-3.

```
VAR
  scratch_tape: (beginning, middle, last) := beginning,
  out_tape: (beginning, middle, last) := last;
    .
    .
    .
```

Figure 4-3. Incorrect Ordinal Variable Type Declaration

In figure 4-3 the ordinal (BEGINNING, MIDDLE, LAST) is defined twice, once for each variable identifier. Using the type declaration (as in figure 4-2), one avoids writing a user declared type over and over.

Type declarations are even more useful with more complicated types. Type declarations facilitate the creation of non-scalar types such as cells (appendix B) and pointers (section 9), the structured types such as sets, strings, arrays, and records (section 8), and the storage types such as sequences and heaps (section 12). Type declarations also help in the declaration of adaptable types and formal types for procedures. These additional uses of the type declaration will be covered in later sections. This section is intended to be an introduction to the use of types.

In summary, type declarations follow all the rules pertaining to scope of identifiers. A type declaration does not occupy storage at execution time, but is used to declare an identifier that stands for a programmer defined type. The type's identifier is typically used in a variable declaration or procedure declaration to refer to the programmer defined type.

5.0 EXPRESSIONS AND THE ASSIGNMENT STATEMENT

5.1 OPERATORS

CYBIL provides five classes (or levels) of operators. Each operator performs an operation on a value or pair of values to produce a result. The following list shows the operator classes in descending order of evaluation precedence.

- NOT operator
- Multiplicative operators
- Sign operators
- Additive operators
- Relational operators

When an expression involves several operators, CYBIL determines the order of evaluation from the order of precedence. For example, if a statement contains multiplicative and additive operators (and no parentheses) the multiplicative operations are performed before the additive operations. Parentheses can be used when necessary to change the normal order of evaluation.

The NOT operator negates a boolean operand; that is, NOT TRUE equals FALSE, and NOT FALSE equals TRUE. NOT can be used only with a boolean operand.

Operators in the other classes perform various operations on other scalar types as well as on types that have not yet been discussed (nonscalar types). The following tables summarize the operators, their meanings, and their operand and result types. Operations on nonscalar types are described in the sections that introduce the types. This section presents detailed descriptions (where necessary) of operations on scalar types only.

The multiplicative operators include multiplication (*), division (DIV), remainder (MOD), and logical and (AND), and are summarized in table 5-1. The logical and operator (AND) results in TRUE if and only if both of its operands are TRUE.

TABLE 5-1. MULTIPLICATIVE OPERATORS

Operator	Meaning	Operands	Result
*	Multiplication	Integer	Integer
DIV	Integer quotient		
MOD	Remainder		
AND	Logical and	Boolean	Boolean

The sign operators include integer identity (+) and integer sign inversion (-), and are summarized in table 5-2.

TABLE 5-2. SIGN OPERATORS

Operator	Meaning	Operand	Result
+	Identity		
-	Negation	Integer	Integer

The additive operators include addition (+), subtraction (-), logical difference (-), inclusive logical or (OR), and exclusive logical or (XOR), and are summarized in table 5-3.

TABLE 5-3. ADDITIVE OPERATORS

Operator	Meaning	Operands	Result
+	Addition		
-	Subtraction	Integer	Integer
OR	Logical or (inclusive)		
XOR	Logical or (exclusive)	Boolean	Boolean
-	Logical difference		

The boolean operators in table 5-3 are defined as follows:

- o Inclusive or (OR) - TRUE if either or both operands are TRUE; FALSE otherwise
- o Exclusive or (XOR) - TRUE if operands are unequal; FALSE otherwise
- o Logical difference (-) - TRUE if left operand is TRUE and right operand is FALSE; FALSE otherwise

The relational operators include less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (=), and not equal to (<>), and are summarized in table 5-4.

TABLE 5-4. RELATIONAL OPERATORS

Operator	Condition	Operands	Result
<	Less than	Both must be same scalar type	Boolean
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
=	Equal to		
<>	Not equal to		

5.2 SCALAR FUNCTIONS

5.2.1 PREDECESSOR AND SUCCESSOR FUNCTIONS (PRED AND SUCC)

The predecessor function determines the predecessor of the scalar argument X. If the predecessor of X does not exist, the program is in error. For example, PRED(7) is the value 6, PRED(0) is the value -1, PRED('B') is the character 'A', PRED(TRUE) is FALSE, and so on.

This function is often used to determine the predecessor of an ordinal identifier.

```
VAR
  x : (in, out, hold, action);
 .
 .
 .
x := hold;
x := PRED(x);
```

Figure 5-1. Predecessor Function (PRED)

In figure 5-1, the ordinal variable X is assigned the ordinal constant HOLD. The statement "x := PRED(x);" determines the predecessor of HOLD (which is OUT) and assigns the value OUT to the variable X.

The successor function (SUCC) returns a result that is the successor of the scalar argument X. It is analogous to the predecessor function described above.

5.2.2 INTEGER CONVERSION FUNCTION (ORD)

This function returns the integer representation of its argument. The argument must be of type character, boolean, or ordinal.

This function is often used to convert a character argument into the ASCII collating sequence ordinal number of the character.

```
VAR
  i : 0 .. 255,
  c : char;
   .
   .
   .
c := 'Z';
i := ORD(c);
```

Figure 5-2. Integer Conversion Function (ORD)

In figure 5-2, the character variable C contains the character 'Z'. The statement "i := ORD(c);" converts the character 'Z' into its integer representation (that is, the value 90 in the ASCII collating sequence) and assigns the value 90 to the variable I. A complete list of the correspondence between ASCII characters and their collating sequence ordinal numbers is given in appendix A.

The integer conversion function also converts an ordinal identifier into its integer equivalent.

```
VAR
  airport : (bos, msp, sfo, dca, dfw),
  value : 0 .. 4;
   .
   .
   .
airport := bos;
value := ORD(airport);
```

Figure 5-3. Integer Conversion Function (ORD)

In figure 5-3, AIRPORT is a variable of type ordinal with five ordinal identifiers declared. Later, the variable AIRPORT is assigned the value BOS. The statement "value := ORD(airport);" converts the current ordinal value of the variable AIRPORT into its integer representation (zero in figure 5-3, because ordinal identifiers are numbered from zero) and assigns the value zero to the variable VALUE. Since ordinal identifiers are numbered from zero, ORD(MSP) has the value one, and ORD(DFW) has the value four.

The integer conversion function also finds the integer equivalent of a boolean (TRUE or FALSE) argument. This is rarely needed, but for completeness it must be added that ORD(FALSE) has the value zero and ORD(TRUE) has the value one.

5.2.3 CHARACTER CONVERSION FUNCTION (CHR)

The character conversion function converts an argument (whose value is in the range $0 \leq X \leq 255$) into a character whose ordinal number in the ASCII collating sequence is X. Refer to appendix A for a complete list of the ASCII collating sequence.

This function is often used to create ASCII characters that have no graphic representation. For example, CHR(7) creates the ASCII code associated with activating an audible signal at a terminal. Seven is the ordinal value of the BEL control function in the ASCII collating sequence.

The CHR function can also be used in constant declarations and variable initialization, as shown in the following examples.

```
CONST                          ! CONST
  nul = CHR(0),                !   nul = 0,
  soh = CHR(1),                !   soh = 1,
  stx = CHR(2),                !   stx = 2,
  etx = CHR(3),                !   etx = 3,
  eot = CHR(4),                !   eot = 4,
  enq = CHR(5),                !   enq = 5,
  ack = CHR(6),                !   ack = 6,
  bel = CHR(7);                !   bel = 7;
                               !
TYPE                           ! TYPE
  ctl_char = nul .. bel;       !   ctl_char = CHR(nul) .. CHR(bel);
                               !
VAR                            ! VAR
  c : ctl_char := nul;         !   c : ctl_char := CHR(nul);
   .                           !    .
   .                           !    .
   .                           !    .
c := bel;                      ! c := CHR(bel);
   .                           !    .
   .                           !    .
   .                           !    .
```

Figure 5-4. Character Conversion Function Examples

The examples in figure 5-4 consist of valid CYBIL statements; both examples produce the same results. In the example on the left, NUL, SOH, and so on, are defined as character constants in the CONST statement. These constants are subsequently used to define a type (subrange of CHAR, NUL .. BEL), initialize a variable (C initially equals NUL), and to assign a new value to C (C := BEL). The TYPE, VAR, and assignment statements in the example on the right in figure 5-4 perform the same functions as their counterparts on the left, but in a slightly different way. Because the identifiers NUL, SOH, STX, and so on, represent integer constants, the CHR function must be used wherever a character value is needed. The method employed in the example on the left is preferred because it enhances readability in a larger portion of the program.

5.2.4 UPPERVALUE AND LOWERVALUE FUNCTIONS

To be supplied.

5.3 EXPRESSIONS

Operators and operands form expressions. The operands must have types suitable for the operator applied to them. For example, the division operator (DIV) must have integer (or integer subrange) left and right operands. It is invalid (a compilation error) to use the division operator to try to divide variables of type integer and boolean.

The order of precedence can sometimes cause unexpected results.

```
VAR
  i, j : 0 .. 100,
  a, b : boolean;
   .
   .
i := 3 + 5 * 4 + 1;              {i := 24}
j := (3 + 5) * (4 + 1);          {j := 40}
 .
 .
a := i < 9;                      {a := FALSE}
b := i < 9 AND j < 4;            {Error}
```

Figure 5-5. Order of Precedence Examples

The sample expressions shown in figure 5-5 provide some insights into the meaning of the order of precedence. The statement "i := 3 + 5 * 4 + 1" results in the value 24 being assigned to I (equivalent to the statement "i := 24"). Since the operators are not the same precedence, the multiplication (5 * 4) is done first, resulting in a value of 20. Then 3 and 1 are added (to 20) giving 24. The value 24 is then assigned to the variable I.

In the statement "j := (3 + 5) * (4 + 1)" parentheses are used to alter the normal order of evaluation. In this statement the left (3 + 5) and right (4 + 1) operands are evaluated before performing the multiplication. The result of the multiplication (40) is assigned to the variable J.

The statement "a := i < 9" involves a relational operator (<) and assigns the value TRUE to boolean variable A if I is less than nine. If I is equal to or greater than nine, the statement assigns FALSE to the boolean variable A.

The statement "b := i < 9 AND j > 4" produces a compilation error. The operator AND is a multiplicative operator (with a higher precedence than < or >) and is evaluated first. However, the left and right operands of the AND operator are 9 and J. The AND operator requires operands of type boolean. To produce results that reflect the intuitive meaning of this statement, it must be rewritten "b := (i < 9) AND (j > 4)". Parentheses force evaluation of (I < 9) followed by evaluation of (J > 4) resulting in two boolean values. Finally, the AND operator is applied to the boolean values. In this example, the parentheses are needed to express the statement correctly and avoid compilation errors.

When the value of certain boolean operations can be determined after evaluation of only the left operand, CYBIL does not evaluate the right operand. Consider the example in figure 5-6.

```
VAR
  k : 0 .. 1000,
  valid : boolean,
  max_char : char;
  .
  .
k :=382;
max_char := '_';
valid := (k <= 255) AND (CHR(k) <= max_char);
```

Figure 5-6. Boolean Operation Evaluation

Since K is greater than 255, the AND operator's left operand is FALSE. There is no need (in this case) to evaluate the right operand: the value assigned to VALID must be FALSE, regardless of the right operand's value. If CYBIL were to evaluate both operands, however, the program would be in error because CHR requires an argument from 0 through 255. CHR(382) would produce an error if its evaluation were attempted.

5.4 ASSIGNMENT STATEMENT

The colon-equal symbol (:=) indicates assignment. Colon-equal is created from the two symbols colon (:) and equals (=) adjacent to one another. Blanks or comments cannot appear between the colon and the equals sign.

One use of the assignment operator was presented in section 3: the initial assignment (initialization) of a value to a variable in a VAR statement. The assignment operator is also used in an assignment statement to assign a value to a variable. The type of the value (or expression) on the right of the assignment operator must be assignable to the type of variable on the left of the assignment operator.

```
VAR
  monthly_salary : 0 .. 4000,
  pay_status : (exempt, non_exempt),
  sex : (male, female);
    .
    .
monthly_salary := 2163;
pay_status := exempt;
sex := female;
```

Figure 5-7. Assignment Examples

The example in figure 5-7 illustrates some assignment statements. The VAR statement (first 4 lines) declares three variable identifiers and their respective types. MONTHLY_SALARY is type subrange of integer (the specific subrange being 0 .. 4000). PAY_STATUS is type ordinal with the programmer declared ordinal constants EXEMPT and NON_EXEMPT. SEX also is an ordinal.

The assignment statement "monthly_salary := 2163;" assigns the value 2163 (type integer) to the variable MONTHLY_SALARY (type subrange of integer). The assignment is valid because type conformability is maintained.

If type conformance is not maintained, the statement is in error. The incorrect assignment may be caught during compilation if constants are invalid. If the error cannot be caught during compilation, it is diagnosed during program execution and causes the program to terminate. The execution time checking can be disabled, but this is not recommended during program development.

```
VAR
  monthly_salary : 0 .. 4000,
  current_raise : 0 .. 200;
    .
    .
monthly_salary := 8000;
current_raise := 150;
monthly_salary := monthly_salary + current_raise;
```

Figure 5-8. Incorrect Assignments

In figure 5-8, the assignment statement "monthly_salary := 8000" is incorrect because the value on the right of the assignment operator (8000) is outside the declared type (subrange of integer 0 .. 4000) of the variable on the left of the assignment operator (MONTHLY_SALARY). CYBIL diagnoses this error during program compilation. Declaring a type for each variable enables the compiler to provide this kind of error checking.

If this error is corrected the program will compile properly, but another error may occur during program execution. An execution-time error occurs if the value of "monthly_salary + current_raise" exceeds 4000. This kind of error cannot be caught during program compilation, but will be diagnosed during program execution.

6.0 ELEMENTARY COMPOUND STATEMENTS

CYBIL provides a variety of statements to control the execution of a program. Among them are the compound statements IF, CASE, BEGIN, WHILE, REPEAT, and FOR. Each of these consists in part of one or more sequences of statements, called statement lists; hence the term compound statements. The precise way in which each compound statement controls execution of its component statement list(s) is the subject of this section and the next.

This section describes the operation of the IF, CASE, and BEGIN statements. These statements each cause (at most) a single execution of a statement list. (WHILE, REPEAT, and FOR provide mechanisms for repeated execution of a statement list and are described in section 7, Repetitive Statements.) The description of the BEGIN statement also introduces the subjects of statement labels and the EXIT statement, both of which are described further in section 7.

6.1 IF STATEMENT

The IF statement causes (or prevents) execution of a statement list depending upon whether a specified condition is true or false. IF statement processing is diagrammed in figure 6-1.

```
             IF
             :
             :                              THEN
       +-----+-----+                       +-----------+
       : Condition : TRUE                  : Statement :
       :   Test    +---------------------->:   List    :
       :    ?      :                       :    1      :
       +-----+-----+                       +------+----+
             :                                    :
       FALSE :                                    :
             :                                    :
             :                    ELSE            :
             :                   +-----------+    :
             :                   : Statement :    :
             +------------------>:   List    :    :
                                 :    2      :    :
                                 +-----+-----+    :
                                       :          :
                                       +--------->:
                                                  :
                                                  v
                                                IFEND
```

Figure 6-1. IF Statement

The IF statement begins with the reserved word IF followed by a condition (a boolean expression). The boolean value of the expression (at the time the IF statement is executed) determines which of two statement lists is executed. If the condition is TRUE, the THEN portion of the IF statement (statement list 1) is executed and the IF statement is terminated (at IFEND). If the condition is FALSE, the ELSE portion of the IF statement is executed (statement list 2). (An alternate form of the IF statement, without the ELSE portion, is described shortly.)

Note that either statement list 1 or statement list 2 is executed as a result of the condition test.

Figure 6-2 illustrates the syntax of the IF statement.

```
VAR
  x, y : integer;
  .
  .
  .
IF x < 0 THEN
  y := x * x;
  x := x - 1;
ELSE
  y := x DIV 2;
  x := x - 2;
IFEND;
  .
  .
```

Figure 6-2. IF Statement Syntax

The IF statement in figure 6-2 consists of seven lines (from IF to IFEND) but it is one statement, even though it includes other statements. In this example, the condition test answers the question, "Is the value of X less than zero?". If the result is TRUE the THEN portion of the IF statement ("y := x * x; x := x - 1;") is executed and the IF statement terminates. The next statement executed is the one following IFEND.

If the result of the condition is FALSE, the ELSE portion of the IF statement is executed ("y := x DIV 2; x := x - 2;") and the IF statement terminates. The next statement executed is the one following IFEND.

An alternate form of the IF statement allows the ELSE to be omitted. This form is diagrammed in figure 6-3.

```
              IF
              :
              :                            THEN
        +-----+-----+                  +-----------+
        : Condition : TRUE             : Statement :
        :   Test    +----------------->:   List    :
        :    ?      :                  :           :
        +-----+-----+                  +-----+-----+
        FALSE :                              :
              :<-----------------------------+
              :
              v
            IFEND
```

Figure 6-3. IF Statement Without ELSE

The short form of the IF statement has only one statement list. If the condition is TRUE, the statement list is executed. If the condition is FALSE, no statements (within the IF-IFEND) are executed.

The short IF statement has many uses. One example is to control the writing (by the program) of debug information, as shown in figure 6-4.

```
CONST
  debug = TRUE;
  .
  .
IF debug THEN
  {Write debug information}
IFEND;
  .
  .
```

Figure 6-4. Short IF Statement Example

In figure 6-4 DEBUG is a boolean constant with the value TRUE. The IF statement tests the value of DEBUG. If it is TRUE, the statements after THEN are executed (represented by the comment {Write debug information}). To disable debug information, the programmer must change the value of DEBUG to FALSE. Then the IF statement would not execute the instructions that write debug information.

The IF statement controlling the production of debug information could be reproduced as often as required in the source program. DEBUG must be as global as necessary to be accessible by all appropriate IF statements.

Sometimes a program requires many nested tests to properly determine which statement list to execute. The THEN or ELSE portion of an IF statement can contain additional (nested) IF statements, as illustrated in figure 6-5.

```
                         IF
                         :                       THEN
                     +---+---+                  +---------+
               FALSE :       :TRUE              :Statement:
              +------:  x=1  :------------------: List    :
              :      :       :                  :    1    :
              :      +-------+                  +----+----+
 ELSE         :                                      :
+-------------:---------------------------------+    :
:             :IF                               :    :
:             :                   THEN          :    :
:         +---+---+              +---------+    :    :
:   FALSE :       : TRUE         :Statement:    :    :
:      +--:  y=1  :--------------: List    :    :    :
:      :  :       :              :    3    :    :    :
:      :  +-------+              +----+----+    :    :
:      :                              :         :    :
:  ELSE:                              :         :    :
: +----+----+                         :         :    :
: :Statement:                         :         :    :
: : List    :                         :         :    :
: :    4    :                         :         :    :
: +----+----+                         :         :    :
:      :                              :         :    :
:      +------------>-+-<-------------+         :    :
:                     :                         :    :
:                     v IFEND                   :    :
+---------------------:-------------------------+    :
                      :                              :
                      : <----------------------------+
                      v
                    IFEND
```

Figure 6-5. Nested IF Statements

In figure 6-5, statement list 1 is executed when X equals 1. Statement list 3 is executed when X is not 1 and Y is 1. Statement list 4 is executed when neither X nor Y is 1. The apparent omission of "statement list 2" is not an oversight. This was done to point out that the second (nested) IF statement <u>is</u> the statement list which constitutes the ELSE portion of the first (outer) IF statement. Translating the example in figure 6-5 into CYBIL statements produces figure 6-6.

```
IF x = 1 THEN
  {Statement list 1}
ELSE
  IF y = 1 THEN
    {Statement list 3}
  ELSE
    {Statement list 4}
  IFEND;
IFEND;
```

Figure 6-6. Nested IF Statements

6.2 CASE STATEMENT

The CASE statement selects one and only one of several statement lists depending upon the value of a scalar expression. One or more IF statements can always replace a CASE statement. In many instances though, the CASE statement formulation represents the intent of the programmer more clearly and may improve program clarity.

Figure 6-8 illustrates the flow of control in a CASE statement.

```
          CASE
            |
     +------+------+
     |Case selector|
     +------+------+
         OF|                  +-----------+
           |= Case spec 1 =  |Statement  |
           |---------------->| List 1    |--->|
           |                 +-----------+    |
           |                                  |
           |                 +-----------+    |
           |= Case spec 2 =  |Statement  |    |
           +---------------->| List 2    |--->|
           |                 +-----------+    |
           |                                  |
           |                 +-----------+    |
           |= Case spec 3 =  |Statement  |    |
           +---------------->| List 3    |--->|
           |  .              +-----------+    |
              .                               |
              .                               |
           |                 +-----------+    |
           |= Case spec n =  |Statement  |    |
           +---------------->| List n    |--->|
           |                 +-----------+    |
           |                                  |
           |                 +-----------+    |
           |ELSE             |Statement  |    |
           +---------------->| List      |--->|
                             +-----------+    |
                                              |
           +----------------------------------+
           |
           v
         CASEND
```

Figure 6-8. CASE Statement

The syntax of the CASE statement is shown in figure 6-9.

```
CASE {Case selector} OF
= {Case specification 1} =
  {Statement list 1}
= {Case specification 2} =
  {Statement list 2}
 .
 .
 .
= {Case specification n} =
  {Statement list n}
ELSE
  {Statement list}
CASEND
```

Figure 6-9. CASE Statement Syntax

The case selector (between the reserved words CASE and OF) determines which of the CASE statement's statement lists is executed; the selector must be a scalar variable or expression. The statement list that is executed is the one that follows the case specification containing the case selector's value at execution time.

A case specification describes what value or values the case selector can have to cause execution of a statement list. A case specification has the following general form:

```
= value_spec, value_spec, ... =
```

Each "value_spec" is either a scalar constant, a scalar expression containing constants only, or a range of constants. For example, the statement list following

```
= 3, 5+1, -2..0, 11..9 =
```

would be executed if the value of the case selector were -2, -1, 0, 3, 6, 9, 10, or 11.

If the value of the case selector does not equal a value specified in any case specification, the statement list following the ELSE is executed. The ELSE clause is, however, optional. If it is omitted and the case selector value does not match a value in a case specification, the program is in error.

The example in figure 6-10 illustrates the CASE statement.

```
VAR
  code : char,
  column : 1 .. 100,
  line_# : 1 .. 60,
  margin : 1 .. 80 := 10;
  .
  .
  .
CASE code OF
= 'p', 'P' =
  column := margin + 4;
  line_# := line_# + 1;
= 'o', 'O' =
  column := column - 1;
= ' ' =
  { Another statement list }
  .
  .
ELSE
  { Default statement list }
CASEND
```

Figure 6-10. CASE Statement Example

The CASE statement in figure 6-10 executes one of several statement lists according to the value of CODE (as determined during program execution). If CODE equals uppercase or lowercase P, the first statement list is executed; if CODE equals uppercase or lowercase O, the second statement list (a single statement in the example) is executed; if CODE equals a space, the third statement list is executed. If code does not match any of the values in the three case specifications, the statement list following ELSE is executed.

The following features of the CASE statement in figure 6-10 should be noted.

- o The scalar type of the case values matches the type of the case selector. In this example, they are type CHAR.
- o The case selector is a variable (or an expression containing variables); the individual cases ('p', 'P', 'o', and so on) are constants (or expressions containing only constants).
- o Only one statement list is executed. No constant appears more than once among all the cases.

6.3 BEGIN_STATEMENT

The BEGIN statement provides a mechanism for the logical grouping of statements. The BEGIN statement is introduced with the reserved word BEGIN and terminated with the reserved word END. Source program statements which, taken as a group, perform some identifiable function are typically placed within a BEGIN statement. The BEGIN statement flowchart is shown in figure 6-11.

```
               BEGIN
                 :
                 v
        +----------------+
        :                :
        : Statement list :
        :                :
        +-------+--------+
                :
                v
               END
```

Figure 6-11. BEGIN Statement Flowchart

The flowchart in figure 6-11 shows that the flow of control in a BEGIN statement is from BEGIN to END. No repetitions occur. The flow of control would be the same (for the statement list) even if the BEGIN statement (BEGIN and END reserved words) were omitted. Declarations (such as variable declarations, type declarations, constant declarations, and so on) are not allowed in a BEGIN statement.

The syntax of the BEGIN statement is shown in figure 6-12.

```
/ label /
  BEGIN
    { Statement list }
   .
   .
   .
  END / label /
```

Figure 6-12. BEGIN Statement Syntax

The label that (optionally) precedes and follows the BEGIN statement is a unique identifier whose construction follows the rules given in section 2. The label following END is optional; if used, it must match the label preceding BEGIN. Labels can add much to the understandability of a program when used wisely. As such, their use is strongly recommended, especially when the labeled statement is many lines long.

Labels cannot, however, be used with all CYBIL statements. They precede (that is, label) only the BEGIN statement and the repetitive statements described in section 7. Reference to labeled statements is made by specifying the label as part of an

EXIT or CYCLE statement only. (The EXIT statement is described at the end of this section; the CYCLE statement is described at the end of section 7.)

An example of a labeled BEGIN statement is shown in figure 6-13.

```
   TYPE
     sizes = (small, medium, large),
     formats = (line, coordinates);

   CONST
     default_size = medium,
     default_format = coordinates;

   VAR
     size : sizes := default_size,
     format : formats := default_format;
    .
    .
    .
 /reset_defaults/
   BEGIN
     size := default_size;
     format := default_format;
   END /reset_defaults/
```

Figure 6-13. BEGIN Statement Example

The description of the BEGIN statement thus far depicts a rather useless statement. The judicious use of blank lines, indentation, and comments serves to group statements and to document their function, perhaps better than the BEGIN statement. The BEGIN statement's utility will increase, however, as additional topics are presented. Specifically,

- o The cross-reference listing (an inventory of identifiers together with other useful information about a CYBIL program) includes label identifiers and their location, easing the search for a small portion of a large program. The cross-reference listing is discussed in detail in a later section of this guide.

- o Execution of a BEGIN statement can be prematurely terminated by an EXIT statement executed within the BEGIN statement.

EXIT Statement

The EXIT statement terminates execution of a BEGIN (or repetitive)* statement. It has no meaning outside such a statement; CYBIL diagnoses such use as a compilation error.

The syntax of the EXIT statement is shown in figure 6-14.

```
EXIT /label/
```

Figure 6-14. EXIT Statement

Execution of an EXIT statement terminates execution of the BEGIN statement with the matching label. The use of labels allows exiting a BEGIN statement from within several levels of nesting. Execution resumes with the statement that follows the exited statement.

The example in figure 6-15 illustrates the BEGIN and EXIT statements.

```
    VAR
      more_args : boolean,
      no_of_args : integer;
     .
     .
    { Determine no_of_args and }
    { actual argument values }

  /process_arguments/
    BEGIN
      more_args := no_of_args > 0;
      IF NOT more_args THEN
        EXIT /process_arguments/;
      IFEND;

      { Process first argument }

      more_args := no_of_args > 1;
      IF NOT more_args THEN
        EXIT /process_arguments/;
      IFEND;

      { Process second argument }

        .
        .
    END /process_arguments/
```

Figure 6-15. BEGIN and EXIT Statements Example

---

* The remainder of this section describes the use of the EXIT statement within a BEGIN statement. Its use within repetitive statements is analogous to this and is discussed further in section 7.

The example in figure 6-15 is a much-simplified representation of the preliminary processing and validation of items in some hypothetical order-dependent argument list. Statements that perform the actual processing of each argument are not shown, but it is assumed that each argument requires some unique processing not required by the others.

The details of the example are (hopefully) self-explanatory. In brief, the BEGIN statement processes arguments until one of the following conditions exists:

- o There are no arguments left to process (MORE_ARGS is FALSE)

- o The BEGIN statement processes the last argument that it must process (the BEGIN statement ends normally)

7.0 REPETITIVE STATEMENTS

Each of the three repetitive statements WHILE, REPEAT, and FOR causes a statement list to be repeated; the different ways in which they perform this task is the main topic of this section.

Like BEGIN, a label can precede a repetitive statement, and an EXIT statement can prematurely terminate execution of a repetitive statement. Additionally, a CYCLE statement can affect a repetitive statement's function in a manner suggested by its name: the repetitive statement is "cycled," thereby repeating its statement list. This brief mention of labels and the EXIT and CYCLE statements is made for two reasons: to introduce topics that are covered in detail later in the section, and to illuminate somewhat the flowcharts that accompany the descriptions of the repetitive statements. Each flowchart, in addition to illustrating the operation of a particular statement, indicates where control is transfered upon execution of an EXIT or CYCLE statement.

7.1 WHILE STATEMENT

The WHILE statement evaluates a condition (boolean expression) prior to executing its statement list. If the condition is false, the statement list is not executed, the WHILE statement ends, and processing continues with the statement after the WHILE statement. (The statement list is not executed even once if the boolean condition is initially FALSE.) If the condition is true, the statement list is executed and the WHILE statement is repeated, beginning with the re-evaluation of the condition following WHILE. The flowchart in figure 7-1 illustrates the operation of the WHILE statement.

```
          WHILE
            |
            |<----------+
            |           |
            v           |
      +-----|-----+     |
FALSE |  Boolean  |     |
 +----| Condition |     |
 |    |     ?     |     |
 |    +-----|-----+     |
 |          | TRUE      |
 |     DO   |           |
 |    +-----+-----+     |
 |    | Statement |     |
 |    |   List    |     |
 |    +-----+-----+     |
 |   CYCLE->|           |
 |          +-----------+
 |
 |
 +----------+
            |
            v
         WHILEND
     EXIT-->|
            |
```

Figure 7-1. WHILE Statement

The reserved word WHILEND terminates the WHILE statement. If an EXIT statement is executed within the WHILE statement, execution continues with whatever follows the WHILE statement (that is, whatever follows WHILEND). If a label precedes the WHILE statement, it can be repeated after WHILEND.

An example of the WHILE statement is shown in figure 7-2.

```
    VAR
      n : 0 .. 84 := 5,
      factorial : integer := 1;
    {Compute factorial of n}
  /compute_factorial/
    WHILE N > 0 DO
      factorial := factorial * n;
      n := n - 1;
    WHILEND /compute_factorial/
```

Figure 7-2. WHILE Statement Example

In figure 7-2, N is declared to be a subrange of the integers 0 .. 84, with an initial value of 5; FACTORIAL is an integer variable with an initial value of 1. N was not declared to be a constant because the program alters the value of N, and constants cannot be altered.

The first operation in the WHILE statement is to determine if N is greater than zero. If N is not greater than zero, the statement list of the WHILE statement is not executed and FACTORIAL remains one (recall that zero factorial equals one). If N is greater than zero, the WHILE statement performs the necessary iterations to compute the proper value of FACTORIAL.

7.2 REPEAT STATEMENT

A unique feature of the REPEAT statement is that the statement list is always executed once upon entering the REPEAT statement. Thereafter, the boolean condition determines if additional repetitions will be performed.

The flowchart in figure 7-3 illustrates the flow of control in a REPEAT statement.

```
                  REPEAT
                     :
   +------------>:
   :                 v
   :            +-----:-----+
   :            : Statement :
   :            :   List    :
   :            +-----:-----+
   :       CYCLE->:
   :                 :
   :        UNTIL :
   :            +-----+-----+
   : FALSE :           :
   +-------:  Boolean  :
            : Condition :
            +-----+-----+
                  : TRUE
          EXIT-->:
                  v
```

Figure 7-3. REPEAT Statement Flowchart

In the flowchart in figure 7-3, note that the statement list is executed once upon entry into the REPEAT statement. After the statement list is executed once, the boolean condition is evaluated. If the condition is true, the REPEAT statement terminates and execution proceeds with the statement after the REPEAT statement (that is, after the UNTIL clause); this is also where execution transfers if an EXIT statement is executed within the REPEAT statement. If the condition is false, the statement list repeats.

The statement list may contain any executable statement including nested REPEAT statements. The REPEAT statement can be read "Repeat the statement list until the boolean condition is true". Unlike the other statements that can be preceded by a label, a label cannot follow a REPEAT statement (that is, cannot follow the UNTIL clause that terminates a REPEAT statement). An example of the REPEAT statement is shown in figure 7-4.

```
    CONST
      limit = 55; {Maximum odd integer}

    VAR
      sum : integer,
      current : 1 .. limit + 2;

    {Find sum of odd integers 1 to limit}

    sum := 0;
    current := 1;
  /sum_odd_integers/
    REPEAT
      sum := sum + current;
      current := current + 2;
    UNTIL current > limit;
```

Figure 7-4. REPEAT Statement Example

The program in figure 7-4 finds the sum of the odd integers 1 to LIMIT. Notice that LIMIT is a constant and the variable items in the program are expressed in terms of this constant. To find the sum of odd integers up to a value other than 55, only the constant declaration need be changed.

7.3 FOR STATEMENT

The FOR statement uses a control variable and programmer-specified initial and final values to control the number of iterations of its statement list. A simple FOR statement is shown in figure 7-5.

```
    VAR
      index : 1 .. 100;
      .
      .
  /for_statement_example/
    FOR index := 1 TO 100 DO
      .
      .
      {Statement list}
      .
      .
    FOREND /for_statement_example/
```

Figure 7-5. FOR Statement Syntax

The FOR statement is introduced with the reserved word FOR. The variable INDEX (from "FOR index := 1 TO 100 DO"), is called the control variable because its value is used to control the number of iterations of the FOR statement. The values 1 and 100 are the initial and final values, respectively. In this example the initial and final values are constants. In general, however, they can be expressions involving scalar variables and constants that are evaluated at execution time. The FOR statement in figure 7-5 would cause its statement list to be executed 100 times.

The end of the FOR statement is designated by the reserved word FOREND. All statements between FOR and FOREND constitute the statement list of the FOR statement. CYBIL places no restriction on the statements comprising the statement list except that no statement can assign a value to the control variable. CYBIL diagnoses such use as an error. The FOR statement can be preceded and followed by a label.

The example in figure 7-5 does not explain all features of the FOR statement. The flowchart in figure 7-6 further explains the operation of a FOR statement.

```
                      FOR
                       :
                       v
             +---------------------+
             :Compute initial value:
             : and assign to temp  :
             +----------+----------+
                        :
                        v
              +-------------------+
              :Compute final value:
              +---------+---------+
                        :
     +--------------->:
     :                  :
     :          +-------+-------+
     :          :               : FALSE
     :          : temp <= final :---------+
     :          :       ?       :         :
     :          +-------+-------+         :
     :                  : TRUE            :
     :                  :                 :
     :       DO         :                 :
     :    +-----------+------------+      :
     :    :control variable := temp:      :
     :    +-----------+------------+      :
     :                  :                 :
     :                  v                 :
     :          +--------------+          :
     :          :Statement list:          :
     :          +------+-------+          :
     :          CYCLE->:                  :
     :                 v                  :
     :         +----------------+         :
     :         :Find successor of:        :
     +---------: temp and assign :        :
               :    to temp      :        :
               +-----------------+        :
                                          :
                        +-----------------+
                        :
                        v
                      FOREND
                  EXIT-->:
```

Figure 7-6. FOR Statement

As shown in figure 7-6, the FOR statement involves many steps. The initial value is evaluated first and assigned to a temporary variable (TEMP in figure 7-6) that the programmer cannot access (it is assigned by the CYBIL compiler). The final value is then determined. Next, a test is made to determine if the value of TEMP (the initial value) is less than or equal to the final value. If the result is false (that is, the value of TEMP is greater than the final value) the FOR statement is terminated. Note that in this case, no assignment is made to the control variable.

If the value of TEMP (the initial value) is less than or equal to the final value, the FOR statement continues and the value of TEMP is assigned to the control variable. The statement list is then executed. The statement list may contain any valid executable statement including additional FOR statements.

Finally, the successor of the value in TEMP is determined and assigned to TEMP. The FOR statement continues by testing to see if the value in TEMP has exceeded the final value. The FOR statement terminates when the value in TEMP exceeds the final value.

When the FOR statement terminates normally, the control variable equals the final value. If an EXIT statement terminates a FOR statement, execution proceeds with whatever follows FOREND and the control variable retains the value it had when the EXIT statement was executed.

The control variable, initial value, and final value need not be type integer as shown in figure 7-5. They may be any scalar type, but they must all be the same type or subranges of the same type. (Note that the successor function determines the next value of TEMP; see the last box of the flowchart in figure 7-6.)

The use of the successor function is more apparent in the example in figure 7-7.

```
VAR
  hardware : (tacks, nails, spikes, bolts, nuts);
  .
  .
FOR hardware := nails TO bolts DO
  .
  .
  {Statement list}
  .
  .
FOREND;
```

Figure 7-7. FOR Statement With Ordinal

In figure 7-7, HARDWARE is an ordinal variable. The permissible ordinal constants for HARDWARE are declared in parentheses. The FOR statement iterates three times. Initially, the variable HARDWARE has the value NAILS. The second iteration is made with the control variable (HARDWARE) set to SPIKES. The third and final iteration is made with the control variable set to BOLTS.

After the FOR statement is completed (after three iterations), the variable HARDWARE will have the value BOLTS.

Another example illustrating the FOR statement is shown in figure 7-8.

```
TYPE
  range = 0 .. 9;
  .
  .
VAR
  index : range,
  init : range := 8,
  final : range := 3;
  .
  .
FOR index := init TO final DO
  .
  .
  {Statement list}
  .
  .
FOREND;
```

Figure 7-8. FOR Statement (No Iterations)

In the FOR statement in figure 7-8 the initial value (INIT) is greater than the final value (FINAL). The flowchart in figure 7-6 shows that the statement list is not executed. The value of the control variable is undefined (never initialized) in this case. This is just one way that undefined variables can be created in a program.

Descending FOR statements are indicated by the use of the reserved word DOWNTO instead of the reserved word TO. In a descending FOR statement, the initial value is normally greater than the final value. The predecessor function is used instead of the successor function to determine the values of the control variable. An example of a descending FOR statement is shown in figure 7-9.

```
VAR
  counter : 831 .. 925;
  .
  .
FOR counter := 900 DOWNTO 850 DO
  .
  .
  {Statement list}
  .
  .
FOREND;
```

Figure 7-9. Descending FOR Statement

CYCLE Statement

The CYCLE statement iterates a repetitive statement in which it is contained. It has no meaning outside such a statement; CYBIL diagnoses such use as a compilation error.

The syntax of the CYCLE statement is shown in figure 7-10.

```
CYCLE /label/
```

Figure 7-10. CYCLE Statement

The CYCLE statement causes the remaining statements in a repetitive statement to be skipped. After skipping, execution continues with whatever function normally follows execution of the statement list. The diagrams in figures 7-1, 7-3, and 7-6 indicate this function for the WHILE, REPEAT, and FOR statements, respectively.

The example in figure 7-11 illustrates the CYCLE statement. (The WHILE statement is used in this example; any repetitive statement can encompass a CYCLE statement, however.)

```
/example/                : /example/
  WHILE x < y DO         :   WHILE x < y DO
    .                    :   /bgn/
    .                    :     BEGIN
    IF c THEN            :       .
      CYCLE /example/    :       .
    IFEND;               :       IF c THEN
    .                    :         EXIT /bgn/
    .                    :       IFEND;
  WHILEND /example/      :       .
                         :       .
                         :     END /bgn/
                         :   WHILEND /example/
```

Figure 7-11. CYCLE Statement Example

In figure 7-11, X and Y are scalar variables, C is a boolean variable. The statement list of the WHILE statement on the left consists of several statements; the statement list of the WHILE statement on the right consists of a single BEGIN statement. Assuming that the statements inside the WHILE statement on the left are the same as the statements inside the BEGIN statement on the right -- except for the IF statements shown -- the two WHILE statements perform identically. (The structure on the right should never be used in practice; it is shown here only to illustrate the operation of the CYCLE statement.)

SUMMARY: Labels, CYCLE, and EXIT

The following points summarize the use of labels and the CYCLE and EXIT statements.

- o Labels can precede (label) only the BEGIN statement and the repetitive statements.

- The reserved word that terminates a BEGIN, WHILE, or FOR statement (END, WHILEND, FOREND) can (and should) be followed by the label of the statement that it terminates. This practice improves program readability especially when many levels of nesting and/or lengthy statements are used.

- A label should be used to identify the function of the statement that it labels whenever possible. This not only makes the program easier to understand, but aids in locating a particular portion of a program via the cross-reference listing.

- The labels on an EXIT or CYCLE statement can specify which of several nested statements is to be cycled or exited.

- The EXIT statement terminates execution of an enclosing BEGIN or repetitive statement; the CYCLE statement causes the remaining statements inside a repetitive statement to be skipped, thereby cycling (iterating) the repetitive statement.

8.0 STRUCTURED TYPES

This section examines data types that consist of collections of components. Unlike scalar types (discussed in sections 3 and 4), the successor and predecessor of a given structured type cannot (in general) be found.

The structured types include array, record, set, and string. Each of these types provides a unique capability for organizing and referencing data.

8.1 ARRAY TYPE

The array type provides random access to its elements, all of which are of the same type. An array is defined in terms of an index and a component type. The general form of an array type definition is shown in figure 8-1.

```
TYPE
  identifier = packing ARRAY [index] OF component type
```

Figure 8-1. ARRAY Type Definition

The type identifier is declared by the programmer. The equal sign indicates an equivalence between the identifier (on the left) and the type declaration (on the right). The packing specification is optional and is used to indicate programmer declared trade-offs between the storage space required to contain the array and the access time needed to reference an array component. The packing specification (PACKED) is explained later in this section. An array type declaration begins with the reserved word ARRAY, includes a specification for the index enclosed in brackets, and uses the reserved word OF to introduce the component type. The programmer must supply an identifier, index, and component type to complete the array type declaration.

The index is restricted to scalar types other than integer (boolean, character, and ordinal) and subranges of scalar types. The component type may be a scalar type (discussed in section 3), a structured type, or a pointer type (discussed in section 9). The component type is the type of all elements comprising the array.

Perhaps the simplest array structure is one that is one dimensional, has integer indexes, and contains integer components. This type of array is illustrated in figure 8-2.

```
       +---------------+
     1 : Integer value :
       +---------------+
     2 : Integer value :
       +---------------+
     3 : Integer value :
       +---------------+
     4 : Integer value :
       +---------------+
               .
               .
               .
       +---------------+
    98 : Integer value :
       +---------------+
    99 : Integer value :
       +---------------+
   100 : Integer value :
       +---------------+
```

Figure 8-2. Array Representation (Simple)

In this example, the indexes are the values 1 to 100. The components are the integer values. This array, in CYBIL syntax, is shown in figure 8-3.

```
TYPE
  simplearray = ARRAY [1 .. 100] OF integer;

VAR
  data_array1, data_array2 : simplearray;
```

Figure 8-3. Array Syntax (Simple)

In figure 8-3, the type declaration (SIMPLEARRAY) identifies the structure of the data (it takes no storage space at execute time). The variables DATA_ARRAY1 and DATA_ARRAY2 are each arrays of type SIMPLEARRAY. Recall (from sections 3 and 4) that variables occupy storage space during program execution. When the variables are automatic, the storage space required to contain the variable exists during execution of the block in which the variable is declared. Type declarations are used only to associate an identifier with a type declaration. Type identifiers never occupy any storage.

The entire array can be referenced by using the array name (for example, DATA_ARRAY1). The only operation permitted on an entire array is assignment (to an array of identical type). Individual components of an array may be accessed by specifying the array name followed by the index of the component enclosed in brackets (for example, DATA_ARRAY2[100]). Examples of both array references and component references are illustrated in figure 8-4.

```
CONST
  limit = 10;

TYPE
  oddtype = ARRAY[1 .. limit] OF 1 .. (limit * 2 - 1);

VAR
  odd_table : oddtype,
  index : 1 .. limit,
  extra_table : oddtype;

FOR index := 1 TO limit DO
  odd_table[index] := index * 2 - 1;
FOREND;
  .
  .
  .
extra_table := odd_table;
```

Figure 8-4. Array References

In figure 8-4, the FOR statement initializes each element of the array ODD_TABLE to an odd integer (via the statement "odd_table[index] := index * 2 - 1;"). A few lines later, the statement "extra_table := odd_table;" performs an array assignment of the array ODD_TABLE to EXTRA_TABLE. Execution time error checking of array indexes is optional and may be selected or deselected using the compile time facilities described in section 14.

8.1.1 ARRAY INITIALIZATION

Array variables are initialized in a fashion similar to the initialization of scalar variables (discussed in section 3). Since an array variable consists of many components, the initialization expression contains many values. The initialization expression for an array variable is enclosed in brackets.

```
CONST
  limit = 3;

TYPE
  oddtype = ARRAY [1 .. limit] OF 1 .. (limit * 2 - 1);

VAR
  base_table : oddtype := [1, 3, 5];
  .
  .
```

Figure 8-5. Array Initialization

In figure 8-5, BASE_TABLE is defined to be type ODDTYPE and is initialized to the values [1, 3, 5]. The initialization is equivalent to BASE_TABLE[1] := 1, BASE_TABLE[2] := 3, and BASE_TABLE[3] := 5.

In the initialization expression, an asterisk may be used to indicate uninitialized components. In figure 8-5, changing the initialization expression to [1, *, 5] would initialize the first and third elements of the array BASE_TABLE to 1 and 5, respectively. The second element of BASE_TABLE is not initialized.

When initialization values are to be repeated, a repetition specification can be used.

```
VAR
  zero_and_one_array : ARRAY[1 .. 100] OF integer
                       := [REP 50 OF 0, REP 50 OF 1];
```

Figure 8-6. Array Initialization (REP)

In figure 8-6, the initialization value "REP 50 OF 0" means fifty repetitions of the value zero. These fifty values of zero are followed by fifty repetitions of the value one. The result of the initialization expression is to initialize the array of 100 elements so that the first fifty elements have the value zero and the second fifty elements have the value one.

In an initialization expression, the repetition specification (REP), the asterisk, and values may be used in any order to represent the initialization for the array.

8.1.2 MULTI-DIMENSIONAL ARRAYS

Multi-dimensional arrays are allowed in CYBIL. There is no arbitrary limit on the number of dimensions. An array with two dimensions is illustrated here because of its simplicity.

A two-dimensional array is thought of as an array whose components are single dimensional arrays. For example, consider the two-dimensional array shown in figure 8-7.

```
                 Column
               1   2   3   4   5
             +---+---+---+---+---+
           1 ! 5 !-10! 2 ! 6 ! 3 !
             +---+---+---+---+---+
           2 ! 4 ! 11! 19! -3! 6 !
     Row     +---+---+---+---+---+
           3 ! 2 ! 1 ! -5! 7 ! 8 !
             +---+---+---+---+---+
           4 ! 3 ! -9! 17! 4 ! 15!
             +---+---+---+---+---+
```

Figure 8-7. Two-Dimensional Array Structure (Conventional)

In figure 8-7, the array structure consists of four rows and five columns. In CYBIL, this structure is thought of as an array of four components where each component is itself an array of five components. This conceptualization is shown in figure 8-8.

```
    +-----------------------+
    :                       :
    : +---+---+---+---+---+ :
  1 : : 5 :-10: 2 : 6 : 3 : :
    : +---+---+---+---+---+ :
    +-----------------------+
    : +---+---+---+---+---+ :
  2 : : 4 : 11: 19: -3: 6 : :
    : +---+---+---+---+---+ :
    +-----------------------+
    : +---+---+---+---+---+ :
  3 : : 2 : 1 : -5: 7 : 8 : :
    : +---+---+---+---+---+ :
    +-----------------------+
    : +---+---+---+---+---+ :
  4 : : 3 : -9: 17: 4 : 15: :
    : +---+---+---+---+---+ :
    :                       :
    +-----------------------+
```

Figure 8-8. Two-Dimensional Array Structure (CYBIL)

When translated into CYBIL syntax the structure shown in figure 8-8 is expressed as shown in figure 8-9.

```
TYPE
  twodim = ARRAY [1 .. 4] OF ARRAY [1 .. 5] OF integer;

VAR
  datatable : twodim;
```

Figure 8-9. Two-Dimensional CYBIL Syntax

Figure 8-9 illustrates one way of defining an array whose component type is type array. Another method of defining this two-dimensional array is shown in figure 8-10.

```
TYPE
  innerarray = ARRAY [1 .. 5] OF integer,
  twodim = ARRAY [1 .. 4] OF innerarray;

VAR
  datatable : twodim,
  alternatetable : ARRAY [1 .. 4] OF innerarray;
```

Figure 8-10. Two-Dimensional CYBIL Syntax

In figure 8-10, the array type TWODIM is declared as a four element array whose components are type INNERARRAY. INNERARRAY is declared to be an array type consisting of five elements of component type integer. The type TWODIM in figure 8-10 is equivalent to the type TWODIM in figure 8-9. Similarly, the variable DATATABLE in figure 8-10 is equivalent to the variable DATATABLE in figure 8-9. The variable ALTERNATETABLE in figure 8-10 is equivalent to the variable DATATABLE in figures 8-9 and 8-10.

8.1.3 INITIALIZING MULTI-DIMENSIONAL ARRAYS

Multi-dimensional array variables can also be initialized. The initialization expression contains left and right brackets enclosing the initial values for each array.

```
TYPE
  twodim = ARRAY [1 .. 4] OF
           ARRAY [1 .. 5] OF integer;

VAR
  datatable : twodim
            := [[5, -10, 2, 6, 3],
                [4, 11, 19, -3, 6],
                [2, 1, -5, 7, 8],
                [3, -9, 17, 4, 15]];
```

Figure 8-11. Array Initialization (Two-Dimensional)

In figure 8-11, the variable declaration for DATATABLE is the CYBIL syntax representation of the structure shown in figure 8-8. Note that each array initialization is enclosed in brackets [and].

8.1.4 REFERENCING MULTI-DIMENSIONAL ARRAYS

Multi-dimensional array references can be constructed to access any required array element or the entire array itself. For example, the identifier DATATABLE (in figure 8-11) refers to the entire two-dimensional array (all 20 integers). The reference DATATABLE[1] refers to the first element of the array DATATABLE which is an array of five integers. The reference DATATABLE[3] [4] refers to the fourth element in the array which is the third element of the variable DATATABLE (initialized to 7 in figure 8-11).

Note that the type of DATATABLE[3] [4] is integer (the value 7 in figure 8-11). DATATABLE[3] is type array of integer ([2, 1, -5, 7, 8] in figure 8-11). DATATABLE is type array of array of integer (that is, a two-dimensional array of integers). The programmer may use any of these references in a program as long as the use of the type is correct.

```
CONST
  limit = 4;

TYPE
  twodim = ARRAY[1 .. limit] OF
           ARRAY[1 .. limit] OF 1 .. limit + limit;

VAR
  datatable : twodim,
  row, column : 1 .. limit;

FOR row := 1 TO limit DO
  FOR column := 1 TO limit DO
    datatable[row] [column] := row + column;
  FOREND;
FOREND;
datatable[4] := datatable[1];
```

Figure 8-12. Two-Dimensional Array Manipulations

In figure 8-12, DATATABLE is a two-dimensional array. The FOR statements reference each element in the two-dimensional array. The statement "datatable[row] [column] := row + column" assigns an integer value (row + column) to an element of the two-dimensional array. At the end of the example the statement "datatable[4] := datatable[1]" assigns the array DATATABLE[1] to the array DATATABLE[4]. This has the effect of making row four in the data structure identical to row one.

8.1.5 PACKING ATTRIBUTE FOR ARRAYS

The packing attribute (PACKED) specifies that storage space for array components is to be conserved at the expense of access time. An unpacked array is mapped onto memory so as to conserve access time at the expense of memory space. When the packing attribute is not specified, the array is unpacked. An inner array does not inherit the packing of the structured variable in which it is contained unless packing of the inner structure is explicitly specified.

```
TYPE
  chardata = ARRAY[1 .. 100] OF 'A' .. 'Z',
  booldata = PACKED ARRAY[1 .. 50] OF boolean;

VAR
  slowchartable : PACKED chardata,
  slowbooltable : booldata,
  fastchartable : chardata;
```

Figure 8-13. Array Packing Attributes

The type and variable declarations in figure 8-13 illustrate methods of using the packing attribute. Note that the attribute can be used in the type declaration or the variable declaration.

In the example in figure 8-13, the programmer specifies that the array SLOWCHARTABLE is packed. This choice minimizes the amount of memory space required for the array SLOWCHARTABLE at the expense of additional access time to reference each element of the array. The array FASTCHARTABLE is not packed, so access time is minimized at the expense of storage space.

8.1.6 ARRAY DATA STRUCTURE EXAMPLES

Arrays are used to represent various kinds of data. The examples that follow examine some interesting array structures and illustrate how these arrays might be implemented in CYBIL.

8.1.6.1 Micro-processor Memory

Occasionally it is necessary to represent the memory of some computer in a program. In this example, the memory of a micro-processor (256 words, 8 bits per word) is represented in CYBIL.

```
CONST
  memsize = 256, {number of words}
  maxword = 0FF(16); {Word size (largest storable value)}

TYPE
  micromem = ARRAY [0 .. memsize - 1]
             OF 0 .. maxword;

VAR
  memory : PACKED micromem;
```

Figure 8-14. Array Structure (Microprocessor Memory)

In the example in figure 8-14, the variable MEMORY is a packed array (type MICROMEM). The type MICROMEM is declared in terms of the two constants MEMSIZE and MAXWORD. Using the declarations in figure 8-14, the example in figure 8-15 illustrates one method of filling the simulated microprocessor memory with ones (turning on all bits in the simulated memory).

```
CONST
  ones = maxword;

VAR
  word : 0 .. memsize - 1;
  .
  .
FOR word := 0 TO memsize - 1 DO
  memory[word] := ones;
FOREND;
```

Figure 8-15. Initializing an Array (Microprocessor Memory)

8.1.6.2 Character Translation

When converting from one character set to another, table lookup is often used. A table of characters is created containing the final (new) characters. The (old) characters to be translated comprise the indexes of the array.

This example converts lowercase characters into uppercase characters.

```
TYPE
  chartbl = ARRAY ['a' .. 'z'] OF 'A' .. 'Z'

VAR
  uppercase : chartbl := ['A', 'B', 'C', 'D', 'E',
                'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
                'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
                'V', 'W', 'X', 'Y', 'Z'],
  testchar : char;
    .
    .
    .
  IF (testchar >= 'a') AND (testchar <= 'z') THEN
    testchar := uppercase[testchar];
  IFEND;
    .
    .
    .
```

Figure 8-16. Character Translation

In the example in figure 8-16, the type CHARTBL denotes an array with indexes of the subrange 'a' .. 'z' and components of the subrange 'A' .. 'Z'. The variable UPPERCASE has the array type CHARTBL and is initialized as shown. TESTCHAR is the character to be translated if necessary. The IF statement determines if the value of TESTCHAR is in the subrange 'a' .. 'z'. If so, the value of TESTCHAR is used as an index into the array UPPERCASE and the contents of the array element is assigned to the variable TESTCHAR, thereby converting a lowercase letter into an uppercase letter.

```
CONST
  ones = maxword;

VAR
  word : 0 .. memsize - 1;
  .
  .
FOR word := 0 TO memsize - 1 DO
  memory[word] := ones;
FOREND;
```

Figure 8-15. Initializing an Array (Microprocessor Memory)

8.1.6.2 Character_Translation

When converting from one character set to another, table lookup is often used. A table of characters is created containing the final (new) characters. The (old) characters to be translated comprise the indexes of the array.

This example converts lowercase characters into uppercase characters.

```
TYPE
  chartbl = ARRAY ['a' .. 'z'] OF 'A' .. 'Z'

VAR
  uppercase : chartbl := ['A', 'B', 'C', 'D', 'E',
              'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
              'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
              'V', 'W', 'X', 'Y', 'Z'],
  testchar : char;
    .
    .
    .
  IF (testchar >= 'a') AND (testchar <= 'z') THEN
    testchar := uppercase[testchar];
  IFEND;
    .
    .
    .
```

Figure 8-16. Character Translation

In the example in figure 8-16, the type CHARTBL denotes an array with indexes of the subrange 'a' .. 'z' and components of the subrange 'A' .. 'Z'. The variable UPPERCASE has the array type CHARTBL and is initialized as shown. TESTCHAR is the character to be translated if necessary. The IF statement determines if the value of TESTCHAR is in the subrange 'a' .. 'z'. If so, the value of TESTCHAR is used as an index into the array UPPERCASE and the contents of the array element is assigned to the variable TESTCHAR, thereby converting a lowercase letter into an uppercase letter.

8.1.6.3 Table_Manipulations

When representing data in the form of an array, the indexes of the array can help clarify the program. This is accomplished using ordinal indexes.

The array in this example contains values that represent the farm production (in thousands of dollars) for various grains by state. The table structure is shown in figure 8-17.

	Corn	Wheat	Oats	Barley
Iowa				
Kansas				
Idaho				
Maine				
Texas				

Figure 8-17. Array Structure (Farm Production)

For program clarity, the array should be defined in terms of the states and crops shown in figure 8-17.

```
TYPE
  value_in_thousands = 0 .. 10000
  grain_type = (corn, wheat, oats, barley),
  farm_states = (iowa, kansas, idaho, maine, texas),
  state_produce_value = ARRAY[farm_states] OF
                     ARRAY[grain_type] OF value_in_thousands;

VAR
  state : farm_states,
  grain : grain_type,
  value_table : state_produce_value,
  value : value_in_thousands;
  .
  .
  .
  value := 0;
  FOR state := iowa TO texas DO
    FOR grain := corn TO barley DO
      value := value + value_table[state] [grain];
    FOREND;
  FOREND;
  .
  .
  .
```

Figure 8-18. Using Ordinals With Arrays

With the declarations shown in figure 8-18, the produce value of OATS in MAINE could be determined by accessing VALUE_TABLE[MAINE] [OATS].

8.2 STRING TYPE

The string data type provides a way of defining and manipulating strings of characters. The use of a string usually implies that the string be treated as a unit (one string). Strings are unique in that they allow the programmer to access any substring (group of characters) of the referenced string. An example of a string declaration is shown in figure 8-19.

```
VAR
  inputline : string (80);
```

Figure 8-19. String Variable Declaration

In the example in figure 8-19, the variable is INPUTLINE, and the type is STRING (80). The length of the string is 80 characters. STRING is a reserved word. Packing attributes are not allowed in a string type definition.

8.2.1 STRING REFERENCES

When the name (identifier) of the string variable is used, it refers to the entire string. For example, INPUTLINE in figure 8-19 refers to the entire string of 80 characters.

Any individual character may be referenced by giving the position of the character in the string in parentheses. For example, INPUTLINE (1) is a reference to the first character of the string INPUTLINE. INPUTLINE (80) is a reference to the last character in the string INPUTLINE.

References can also be made to a portion of a string variable, called substring references. The format of a substring reference is as follows:

string identifier (starting position, length)

For example, INPUTLINE (1,10) refers to the substring starting at position one and having a length of ten characters. The length parameter of the substring reference can be an asterisk indicating that the substring extends to the end of the string. For example, INPUTLINE (75,*) is a substring consisting of the last six characters in the string INPUTLINE. Reference to an entire string can be made in a number of ways. For example, INPUTLINE, INPUTLINE (1,80), and INPUTLINE (1,*) all refer to the entire 80 character string INPUTLINE declared in figure 8-19.

String constants are denoted by enclosing the characters comprising the string in apostrophes. For example, 'To be' is a string constant whose length is five. CYBIL does not, however, allow substring references of string constants. The example in figure 8-20 illustrates this.

```
    CONST
      strcon = 'inviolate';

    VAR
      strvar : string (9),
      shortstr : string (5);

{1}  shortstr := strcon (3, 5);    {Invalid substring reference}
{2}  strvar := strcon;             {Valid                      }
{3}  shortstr := strvar (3, 5);    {      string               }
{4}  shortstr := 'viola';          {             assignments   }
```

Figure 8-20. String Constants, Invalid Substring

The presumed object of the program segment in figure 8-20 is to assign the string 'viola' to the string variable SHORTSTR. The first assignment statement contains an invalid substring reference, because STRCON is a string constant. To reference the third through seventh characters of STRCON an intermediate string variable must be used. The second and third assignment statements illustrate this process. Of course, the apostrophe-delimited string constant may also be assigned to a string variable, as shown in the fourth assignment statement. String assignment is described further in the following paragraphs.

8.2.2 STRING ASSIGNMENTS

Assignments to string variables operate under rules that relax the requirement for strict type equivalence of destination variable and value. Briefly, a character, string, or substring value can be assigned to a substring, string variable, or character variable. If the lengths of the value and the destination variable are different, the length of the value is adjusted (truncated or extended) to match the length of the destination variable. CYBIL truncates a string by removing characters from the right. CYBIL extends a string (or character) by appending blanks on the right. The examples in figure 8-21 illustrate these string assignment concepts.

```
VAR
  ch : char,
  s8 : string(8),
  s9 : string(9);
s9 := 'fourscore';
s8 := s9(5,*);                   { 'score   ' }
ch := s8;                        { 's'        }
s8(7, *) := '345';               { 'score 34' }
s8(6, 2) := ch;                  { 'scores 4' }
```

Figure 8-21. String Assignments

The comments in figure 8-21 indicate the value of the destination variable after each assignment statement.

When substrings of the same string variable are involved on both sides of the assignment operator, the substrings must not overlap; the results of such an assignment are undefined. For example, using the variables in figure 8-21, the statement

```
s9(1, 3) := s9(2, 3)
```

is in error; CYBIL issues a diagnostic message during compilation. If, however, the substrings involved are defined in terms of variables, the legality of the assignment cannot be determined until the statement is executed. For example, the statement

```
s9(1, 3) := s9(start, 3)
```

(where START is an integer variable) is in error if START has a value less than 4. In such a case the results are unpredictable.

8.2.3 ARRAYS OF STRINGS

An array of strings is commonly used to construct a table of names. Each component of the array is defined to be a string of the necessary length.

```
CONST
  tablelength = 29, {Max names in table}
  stringlength = 13; {Max chars per name}

VAR
  nametbl : ARRAY [1 .. tablelength] OF
              STRING (stringlength);
```

Figure 8-22. Array of Strings

In figure 8-22, the array NAMETBL consists of 29 strings of 13 characters each. The identifier NAMETBL refers to the entire array of strings. An individual string of characters is referenced as NAMETBL[I], where I is a value in the subrange 1 .. 29. It is also possible to reference any substring in the array. For example, NAMETBL[29] (13) := 'Z', assigns the letter 'Z' to the last character in the last string of the array of strings.

8.2.4 STRING COMPARISON

To be supplied

8.2.5 STRING INITIALIZATION

A string variable is initialized by specifying a string constant after the string variable declaration.

```
CONST
  str1 = 'ABCDE';

VAR
  strvar : STRING (6) := str1,
  strs : STRING (3) := 'XYZ',
  neatstr : STRING (7) := str1 CAT 'QRS';
```

Figure 8-24. String Initialization

Figure 8-24 illustrates various methods of declaring and initializing strings. The constant declaration declares the identifier STR1 to be a string of five characters ('ABCDE'). When the lengths of the string variable and the constant are unequal, the initialization behaves like string assignment: the string constant is truncated or extended on the right to fit the length of the string variable. In figure 8-24, STRVAR is initialized to 'ABCDE '; CYBIL adds a blank to the value of the STR1 to form a six-character string. NEATSTR is initialized to 'ABCDEQR'; CYBIL deletes the S to form a seven-character string.

8.3 RECORD_TYPE

The data structures discussed so far have consisted of homogeneous components. That is, all the elements have been the same type. The record type allows for the creation of a data structure (called a record) that contains nonhomogeneous components. These components are called fields.

```
Field          Type

               +--------------+
surname        |string (11)   |
age            |0 .. 115      |
married        |boolean       |
sex            |(male, female)|
fingers        |0 .. 11       |
               +--------------+
```

Figure 8-25. Record Concepts

As shown in figure 8-25 above, a record consists of many fields. Each field is associated with a type and has a unique field identifier (SURNAME, AGE, and so on). The field identifiers must be unique within the record. The reserved words RECORD and RECEND define the beginning and end of the record definition.

```
TYPE
  personal = RECORD
               surname : string (11),
               age : 0 .. 115,
               married : boolean,
               sex : (male, female),
               fingers : 0 .. 11,
             RECEND;

VAR
  recvar : personal,
  recarray : array [1 .. 100] OF personal;
```

Figure 8-26. Record Syntax

The type PERSONAL in figure 8-26 is a record with five fields. Each field has a field identifier followed by a colon and a field type. The record itself is bounded by the reserved words RECORD and RECEND.

The variable RECVAR is defined to be type PERSONAL. CYBIL allocates to this variable sufficient storage space to contain all the declared fields. An array of records can be created to allow access to many occurrences of the record. An array of records is illustrated with the variable RECARRAY in figure 8-26.

8.3.1 RECORD REFERENCES

In figure 8-26, RECVAR identifies a record of type PERSONAL. Whenever the identifier RECVAR is used, it refers to the entire record. A record can be assigned to another record of the same type and two records of the same type can be compared for equality or inequality. No other operations are permitted on entire records.

A field of a record is referenced by using the record identifier followed by a period and the field name. For example, RECVAR.SURNAME refers to the field SURNAME (STRING (11)) in the record variable RECVAR (refer to figure 8-26). The first five characters of the field SURNAME in the variable RECVAR are referenced RECVAR.SURNAME (1,5).

Consider the array of records RECARRAY in figure 8-26. The reference RECARRAY[51] refers to the 51st record in the array. RECARRAY[51].SURNAME refers to the field SURNAME in the 51st record in the array RECARRAY. Similarly, RECARRAY[51].SURNAME (6,*) refers to the last six-character substring in the field SURNAME in the 51st record of the array RECARRAY.

8.3.2 PACKING AND ALIGNMENT

Packing and alignment attributes are used to specify storage space versus access time trade offs for fields of records. Fields of packed records are mapped onto storage so as to conserve storage space at the expense of execution (access) time. Regardless of packing, aligned fields are mapped with storage so as to be directly addressable. When a field is made directly addressable, the field begins on an addressable boundary to facilitate rapid access to the field. Records themselves (the collection of fields) are always aligned unless they are unaligned fields of a packed structure.

Record packing is specified with the reserved word PACKED. Alignment of fields is specified with the reserved word ALIGNED.

```
TYPE
  data = PACKED RECORD
           name : string (11),
           addr : ALIGNED [O MOD 8] string (31),
           age : 0 .. 115,
           grades : array [1 .. 10] OF 'A' .. 'F'
         RECEND;

VAR
  class_data : array [1 .. 30] OF data;
```

Figure 8-27. Packing and Alignment (Record)

In figure 8-27, DATA is defined to be a packed record type. This means that the fields of this record will be structured to reduce storage space at the expense of access time. The field ADDR is aligned, however. ADDR will begin on an addressable boundary to facilitate rapid access to this field at the expense of storage space. The material following the ALIGNED keyword -- [O MOD 8] -- specifies how the field is to be aligned. Refer to appendix B, Representation Dependent Features, for further information.

8.3.3 VARIANT RECORDS

Tables for system software often require a record with fixed information (or fields) followed by variable (variant) fields. Figure 8-28 shows this pictorially.

```
          +- +------------+          +---------+       --+
          :  :            :          : integer :   A     :
  Fixed  <:  :    name    :      +-->:---------:         :> Short
  Part    :  :            :      :   : boolean :   B     :  Form
          +- :------------:      :   +---------+       --+
             :    type    :      :
          +- :------------:      :
          :  :            :------+
Variant  <:  : attributes :------+   +---------+       --+
Part      :  :            :      :   : ordinal :   C     :
          +- +------------+      :   :---------:         :
                 Record          +-->: boolean :   D     :> Long
                                     :---------:         :  Form
                                     : integer :   E     :
                                     +---------+       --+
```

Figure 8-28. Variant Record Concept

Figure 8-28 shows a record consisting of two major parts: a name and a set of attributes. Depending upon the type of the record, the attributes portion of the record can be one of two forms (the short form or the long form).

In any one occurrence of the record, only one of the two forms will exist. However, some occurrences of the record may be the short form and others may be the long form.

This kind of structure is declared in CYBIL as shown in figure 8-29.

```
TYPE
  form = (short, long),
  color = (red, green, blue),
  rectype = record
            name : string (11)

            CASE t : form OF
            =short= a : integer,
                    b : boolean,
            =long=  c : color
                    d : boolean,
                    e : integer,
            CASEND
            RECEND;

VAR onerec, tworec : rectype;
```

Figure 8-29. CYBIL Variant Record Syntax

The record structure declared by the example in figure 8-29 will be one of the two illustrated in figure 8-30.

```
               ---------------              ---------------
      name     ! string (11) !       name   ! string (11) !
               !-------------!              !-------------!
         t     ! short       !          t   ! long        !
               !-------------!              !-------------!
         a     ! integer     !          c   ! color       !
               !-------------!              !-------------!
         b     ! boolean     !          d   ! boolean     !
               ---------------              !-------------!
                                        e   ! integer     !
                Short variant               ---------------

                                              Long variant
```

Figure 8-30. Variant Record Layout

As illustrated in figure 8-30, there are two forms (or variants) for the record (LONG or SHORT). The identifier T (the tag field) is included in the record itself.

Information is stored into the record variable one field at a time. To declare which variant is being used, a value (LONG or SHORT) is assigned to the tag field T. This is illustrated for each record type (variant) in figure 8-31, which assumes the type declarations made in figure 8-29.

```
 VAR onerec, tworec : rectype;
  .
  .
  .
{Initialize onerec as short record}
 onerec.name := 'IDENTIFIER1';
 onerec.t := short;
 onerec.a := 6;
 onerec.b := TRUE;

{Initialize tworec as long record}
 tworec.name := 'IDENTIFIER2';
 tworec.t := long;
 tworec.c := blue;
 tworec.d := FALSE;
 tworec.e := 5;
  .
  .
  .
```

Figure 8-31. Variant Record Initialization

Since the tag field of the record is stored in the record, one can determine at execution time which variant is contained in the record.

8.4 SETS

The CYBIL notion of a set follows closely the mathematical concept of a set; that is, a set is an accumulation of elements (members). In CYBIL, the elements can be ordinal values (identifiers), characters, integers, or boolean values. The set members have no order; one cannot say that some member is the first member in the set. However, it is possible to place members in a set, delete members from a set, and determine whether a given member exists in a set. One can create both empty and full sets.

8.4.1 SET DECLARATION

Figure 8-32 illustrates the kinds of sets allowed and the syntax for declaring sets.

```
TYPE
  a = set OF (red, green, blue)
  b = set OF 0 .. 17,
  c = set OF 'A' .. 'Z';

VAR
  ordset : a,
  intset : b,
  vowelset : c;
```

Figure 8-32. Declaration of Sets

In figure 8-32 type A denotes a set of the ordinal identifiers (RED, GREEN, BLUE). A variable of type A (ORDSET in figure 8-32) is a set variable and, in this example, can contain up to three members. If ORDSET is empty then it contains no members. If ORDSET is full then the members RED, GREEN, and BLUE are all present in the set.

8.4.2 SET REFERENCES

A set variable takes on a value in the same manner as variables of other types: by assignment or by initialization. The two methods employ slightly different mechanisms for specifying (or constructing) the set value to be assigned to the set variable; therefore, each is described separately, beginning with assignment statements involving set variables.

When assigning a value to a variable of type set, the value assigned (what appears to the right of the assignment operator) must have the same type as the variable. The value can be any expression containing set variables and/or set value constructors (described shortly), as long as all set variables and set value constructors are of the same type.

8.4.2.1 Set Value Constructors

A set value constructor denotes (constructs) a set through an explicit itemization of the elements to be included in the set. Its general format is as follows:

    $set type identifier[list of set elements]

The set elements, enclosed in brackets, are separated by commas. Each must be an expression (containing variables and/or constants) whose type matches the component type of the set. The examples in figure 8-33 illustrate set value constructors.

```
VAR
  i1,
  i2 : 0 .. 17;
 .
 .
i1 := 7;
i2 := i1;
    intset := $b[i1];                                  { 7         }
    intset := $b[i1, i2];                              { 7         }
    intset := $b[i1, i1 + i2, i1 - 3];                 { 7, 14, 4  }
    intset := $b[i1 - i2];                             { 0         }
    intset := $b[];                                    { empty set }
    ordset := $a[blue, PRED (green)];                  { blue, red }
    vowelset := $c['A', 'E', 'I', 'O', 'U'];
```

Figure 8-33. Set Value Constructors

Variables I1 and I2 are subrange of integer type; all other identifiers in figure 8-33 retain the meaning associated with them by the declarations in figure 8-32. Comments appearing to the right of the assignment statements in figure 8-33 indicate the elements of the set variable after each assignment. Note that the empty set is denoted by a set value constructor with no values specified within the brackets. (Methods for creating a full set are discussed in the paragraphs describing set operations.)

8.4.2.2 Set Initialization

Set variable initialization is similar to array and record initialization: the elements comprising the initial set value are listed between brackets and separated from the variable type declaration by := as shown in figure 8-34.

```
VAR
   set_var : set OF 0 .. 15 := [3, 7, 8-7, 4];
```

Figure 8-34. Set Variable Initialization

Set variable initialization is also similar to the set value constructor described earlier, with the following differences:

- o The list of set elements must be constants or expressions involving constants only.
- o No set type identifier precedes the bracketed list of set elements because the set type is specified as part of the variable declaration.

The variable declaration in figure 8-34 does not employ a type identifier (defined in a TYPE statement) to denote the type of SET_VAR. Instead, its type is explicitly defined by "set OF 0 .. 15". Defining a set variable in this way is valid but, without a set type identifier declared, it can restrict future assignments to the variable to expressions without set value constructors.

8.4.3 SET_OPERATIONS

CYBIL provides two groups of operators that operate on sets. Members of the first group operate on set operands and produce set results. These operators are summarized in table 8-1. The second group comprises five relational operators that produce boolean results. These operators are summarized in table 8-2.

TABLE 8-1. SET-VALUED OPERATORS

Operator	Name	Result Definition	Evaluation Precedence
*	Set intersection	The set consisting of all elements common to the two operand sets	1 (highest)
-	Set complement (single operand)	The set of all elements of the base type not in the specified operand set	2
+	Set union	The set consisting of all elements of both operand sets	3
XOR	Symmetric difference	The set consisting of all elements contained in either set but not in both sets	3
-	Set difference (two operands)	The set consisting of all elements of the left operand that are not also elements of the right operand	3

TABLE 8-2. RELATIONAL SET OPERATORS

Operator	Tests for	Sample Expression	Result Definition
=	Identity	set1 = set2	TRUE if all members of set1 are in set2 and all members of set2 are in set1, or if set1 and set2 are empty; FALSE otherwise
<>	Inequality	set1 <> set2	FALSE if set1=set2 is TRUE; TRUE otherwise
<=	Containment (left operand contained in right operand)	set1 <= set2	TRUE if all members of set1 are members of set2, or if set1 is empty; FALSE otherwise
>=	Containment (right operand contained in left operand)	set1 >= set2	TRUE if set2<=set1 is true; FALSE otherwise
IN	Set membership	scalar IN set1	TRUE if scalar value is a member of set1; FALSE otherwise

CYBIL performs the operations in table 8-1 according to the precedence shown (and before relational operations) when an expression involves operators of different precedence.

The operands used with each of the operators in tables 8-1 and 8-2 (except IN, which is described shortly) must be of the same type. The result type of the operations in table 8-1 is the same as that of the operand(s) involved. An operand can be a set value constructor, a set variable, or a set-valued expression.

The IN operator tests whether a value is a member of a set. The type of the value and the component type of the set must be the same, or one must be a subrange of the other. Each can also be a subrange of the same type. If the value is outside the subrange that defines the set's component type when the IN test is performed, the boolean result is FALSE. Set operations are illustrated in figures 8-35 and 8-36.

```
            TYPE
              toe = 1 .. 5,
              toes = set OF toe;

            VAR
              lft : toes := [2, 4, 5],
              rft : toes := [1, 2, 3, 4],
              ft : toes;
             .
             .
{  1 }   ft := lft * rft;        {       2 4 }
{  2 }   ft := - lft;            {       1 3 }
{  3 }   ft := lft + rft;        { 1 2 3 4 5 }
{  4 }   ft := lft XOR rft;      {     1 3 5 }
{  5 }   ft := lft - rft;        {         5 }

{  6 }   ft := - $toes[];        { 1 2 3 4 5 }

{  7 }   ft := - lft * rft;      {     1 3 5 }
{  8 }   ft := - (lft * rft);    {     1 3 5 }
{  9 }   ft := ( - lft) * rft;   {       1 3 }
{ 10 }   ft := - lft + rft;      {   1 2 3 4 }
{ 11 }   ft := ( - lft) + rft;   {   1 2 3 4 }
{ 12 }   ft := - (lft + rft);    { Empty set }
```

Figure 8-35. Set-valued Operations

The following discussion uses the reference numbers that appear to the left of the assignment statements in figure 8-35. To the right of each valid assignment statement is a comment that indicates the value(s) assigned to the result variable.

Statements 1 through 5 illustrate the set-valued operators intersection, complementation, union, symmetric difference, and set difference, respectively. These operators are defined in table 8-1.

Statement 6 shows a straightforward method of denoting a full set: by complementing an empty set. A set value constructor listing all elements of a set's component type also produces a full set, but diminishes program readability. Denoting a full set by itemizing the set contents could also increase the program maintenance effort: if the component type of a set were changed (by addition or removal of elements), any reference to a full set of this type would require modification if its elements were explicitly listed.

The next six statements in figure 8-35 illustrate the way in which CYBIL evaluates set-valued expressions without parentheses. The first group (statements 7, 8, and 9) involves set intersection, the second involves set union. The first statement in each group (statements 7 and 10) has no parentheses; the second statement in each group is equivalent to the first, but includes parentheses. The third statement in each group illustrates the consequences of incorrectly applying the precedence rules for set operators. Because set complementation is performed after set intersection, but before set union, it is good practice to use parentheses to determine the evaluation order of a set-valued expression. This practice not only improves readability, but helps prevent unintentional misevaluation of an expression.

The statements in figure 8-36 demonstrate some fundamental properties of sets and contrast the IN operator with the containment operator. As before, the following discussion uses the numbers appearing to the left of each statement for reference.

```
                TYPE
                  toe = 1 .. 5,
                  toes = set OF toe;

                VAR
                  lft : toes := [2, 4, 5],
                  rft : toes := [1, 2, 3, 4],
                  ft : toes,
                  t3 : toe := 3,
                  t6 : integer := 6,
                  b : boolean;
                 .
                 .
        { 21 }  b := $toes[] = lft XOR lft; {  TRUE }
        { 22 }  b := $toes[] = rft - rft;   {  TRUE }
        { 23 }  b := $toes[1, 4] <= rft;    {  TRUE }
        { 24 }  b := lft >= $toes[];        {  TRUE }

        { 25 }  b := t3 IN rft;             {  TRUE }
        { 26 }  b := $toes[t3] <= rft;      {  TRUE }
        { 27 }  b := 6 IN lft;              { FALSE }
        { 28 }  b := t6 IN lft;             { FALSE }
        { 29 }  b := $toes[t6] <= lft;      { Error }
```

Figure 8-36. Relational Set Operations

Statements 22 through 24 illustrate two basic boolean operations on sets, identity and containment. (Inequality is omitted as it derives its definition from set identity; set membership is illustrated in statements 25 through 29.) Statements 21, 22, and 24 also demonstrate some properties of the empty set: the symmetric difference or set difference of two identical sets is the empty set, and the empty set is a subset of (contained in) any set.

The remaining statements in figure 8-36 illustrate the IN operator. Statements 25 and 26 are functionally equivalent: the value assigned in each case depends on whether the value of variable T3 is a member of RFT. (In this example, T3's value is in RFT, so the result is TRUE.) As will be shown shortly, the equivalence of these two statements is due not only to their construction, but also to the fact that T3's type matches the component type of RFT's type (both are TOE). Although the two statements are equivalent, the form used in statement 25 is recommended, as its meaning is more immediately recognizable.

Statements 27 and 28 show the result of testing for set membership (via IN) when the scalar's type differs from the set's component type with respect to subrange (that is, the parent type -- integer -- is the same, but the subranges differ). If the scalar value is outside the subrange of the set's component type, the expression is FALSE; otherwise the expression's value is dependent on the scalar's membership in the set.

Statements 28 and 29 have the same equivalence in construction as statements 25 and 26, but are not equivalent in meaning. T6's type is integer and can therefore take on values outside the subrange comprising LFT's component type. The set value constructor in statement 29 is invalid whenever the value of T6 is not of the same type (subrange) as LFT's component type; therefore, statement 29 is in error, because T6's value is 6.

9.0 PROCEDURES

The essence of a procedure is the association of an identifier with a statement list such that specifying the identifier causes the execution of the statement list. The association of the statement list with the identifier constitutes the procedure declaration; specification of the identifier (the procedure name) as an element of a statement list constitutes a procedure call statement. Procedures (when used properly) subdivide a large program into manageable tasks, thereby making plain the program's overall structure.

9.1 DECLARATION AND USE

The most general form of a procedure is shown in figure 9-1.

```
PROCEDURE [attributes] procedure name (parameter list);
    .
    .
    { Procedure body }
    .
    .
PROCEND procedure name
```

Figure 9-1. General Structure of a Procedure

In figure 9-1, the reserved word PROCEDURE introduces the procedure, optional attributes specify how the procedure is to be used, the procedure name identifies the procedure, and the optional parameter list specifies arguments passed to or returned by the procedure. The reserved word PROCEND and (optionally) the procedure name indicate the end of a procedure declaration. The procedure body contains the statement list (and other elements that will be discussed shortly) that is executed when the procedure is called. Without its optional parts, the structure illustrated in figure 9-1 has the simple form shown in figure 9-2.

```
PROCEDURE procedure name;
  .
  .
  {Statement list}
  .
  .
PROCEND procedure name
```

Figure 9-2. Simplified Procedure Format

(The procedure name following PROCEND, while not required, improves a program's readability and is highly recommended. For this reason it is included in figure 9-2 and in all other procedure declarations illustrated in this guide.)

Procedures are declared (like variables, constants, and so on) before the statement list in which they are used. Figure 9-3 illustrates the relationship between procedure declaration and use.

```
                          PROCEDURE do_it;
                          ----
                          ---- { Procedure body }
                          ----
                          PROCEND do_it;

                          VAR
                            condition : boolean;
Execution
Begins here -------->     .
                          .
                  {1}     do_it;
                          .
                  {2}     IF condition THEN
                            do_it;
                          IFEND;
                          .
```

Figure 9-3. Execution Flow

The program segment in figure 9-3 shows a procedure declaration and a variable declaration. When the statement list that follows these declarations is executed, the procedure call statement at {1} initiates execution of procedure DO_IT. When procedure DO_IT completes, execution resumes with the statement following the procedure call at {1}. If CONDITION is TRUE when the IF statement at {2} is executed, DO_IT is called again. In this case execution resumes with whatever follows the IF statement (that is, whatever follows IFEND).

Procedures also provide the ability of shielding and sharing variables. Variables can be declared inside a procedure and are thus local variables. Their scope is the procedure (block) in which they are defined. This concept is illustrated in figure 9-4.

```
PROCEDURE outer;
  VAR
    i,
    out_var : integer;

    PROCEDURE inner;
      VAR
        i,
        in_var : integer;
        .
        .                { inner statement list }
        .
    PROCEND inner;
    .
    .                { outer statement list }
    .
PROCEND outer;
```

Figure 9-4. Shielding and Sharing Variables

In figure 9-4 variable OUT_VAR is local to procedure OUTER. Since procedure INNER is contained in procedure OUTER, variable OUT_VAR can also be referenced in procedure INNER. Variable OUT_VAR is global to procedure INNER. Similarly, variable IN_VAR declared in procedure INNER is local to procedure INNER and cannot be referenced in procedure OUTER.

Two distinct variables are denoted by the identifier I. One variable, I declared in procedure OUTER, is local to procedure OUTER and global to procedure INNER. The other variable I is declared in procedure INNER and denotes a local variable that can be referenced only in procedure INNER. This second (inner) declaration of I supersedes the declaration of I in procedure OUTER. Thus the variable I declared in procedure OUTER cannot be referenced within procedure INNER. Table 9-1 summarizes these points.

TABLE 9-1. SCOPE OF IDENTIFIERS EXAMPLE

Variable Identifier	Declared in procedure	Can variable be referenced in procedure OUTER?	Can variable be referenced in procedure INNER?	Scope of the variable identifier
I	OUTER	Yes	No	Procedure OUTER
OUT_VAR	OUTER	Yes	Yes	Procedures OUTER and INNER
I	INNER	No	Yes	Procedure INNER
IN_VAR	INNER	No	Yes	Procedure INNER

9.2 NESTED PROCEDURES

When a procedure declaration is nested (that is, a procedure is declared within another procedure declaration), the nested procedure is shielded just as any other declaration within the outer procedure. The inner procedure cannot be called from outside the outer procedure. For example, consider the program structure in figure 9-5.

```
MODULE nested_procs;
                                                          --+
  PROCEDURE compute;                                        !
    VAR                                                     !  c
      .                                         --+         !  o
      .                                           !         !  m
                                                  !         !  p
    PROCEDURE test;                               !  t      !  u
                                                  !  e      !  t
      VAR                                         !  s      !  e
        .                                         !  t      !
        .                                         !         !
     { Executable statements }                    !         !
    PROCEND test;                               --+         !
                                                            !
   { Executable statements }                                !
  PROCEND compute;                                          !
                                                          --+
  PROCEDURE main;                                         --+
    .                                                       !
    .                                                       !  m
    compute  ;                                              !  a
    .                                                       !  i
    .                                                       !  n
  PROCEND main;                                             !
                                                          --+
MODEND nested_procs;
```

Figure 9-5. Nested Procedures

In figure 9-5 procedures MAIN and COMPUTE are at the outermost level (that is, their declaration is not contained within another procedure declaration). Procedure TEST is nested inside COMPUTE. This structure provides shielding (protection) for procedure TEST. With this structure, procedure MAIN can call procedure COMPUTE, but procedure MAIN cannot call procedure TEST. Only COMPUTE can call TEST. So TEST can rely on COMPUTE to validate variables and perform other computations which might be necessary for the correct execution of TEST.

9.3 PARAMETERS

A procedure's effectiveness depends on its ability to affect, or be affected by, variables that are not local to it. One method by which a procedure can do this was described earlier: a procedure can reference variables global to it, as procedure INNER could with variable OUT_VAR in figure 9-4. A procedure can also interact with its environment via parameters.

9.3.1 FORMAL AND ACTUAL PARAMETERS

Parameters are an optional part of a procedure declaration. They are specified after the procedure identifier, enclosed in parentheses (see figure 9-1). Parameters in the procedure declaration (called formal parameters) correspond to actual parameter values specified on the procedure call statement. This correspondence is illustrated in figure 9-6.

```
TYPE
  days = 0 .. 6,
  total_array = array[days] of integer;

VAR
  first_day : days,
  date : integer,
  tot01 : total_array;

PROCEDURE count_days (     first : days,
                           day_# : integer);
  VAR
    day : days;

  day := (first + day_#) MOD 7;
  tot01[day] := tot01[day] + 1;

PROCEND count_days;

date := 193;
first_day := 0;

count_days (2, date);
count_days (first_day, date + 3);
count_days (first_day, 13);
```

Figure 9-6. Elementary Parameter Passing

The parameter list for procedure COUNT_DAYS provides several pieces of information.

- o It specifies the identifiers that denote the procedure's formal parameters. These identifiers are like read-only variable identifiers whose scope is the procedure itself (COUNT_DAYS in figure 9-6).

- o It specifies the type of each parameter. FIRST denotes a variable of type DAYS (subrange of integer, 0 to 6); DAY_# denotes a variable of type integer. Actual parameter values specified on a statement that calls COUNT_DAYS must conform to these types.

- o It establishes the number and order of parameters for the procedure. Thus, a statement calling COUNT_DAYS (as shown in figure 9-6) specifies two values as actual arguments: the first value is assigned to FIRST, the second is assigned to DAY_# when COUNT_DAYS is called.

The formal parameters in figure 9-6 have the same scope as local variables, but cannot receive assignments; they can appear only where expressions are permitted (to the right of an assignment operator, as an array subscript, in a substring reference, and so on). Each actual parameter specified in the procedure call statements in figure 9-6 can be any expression as long as its value is of the same type as the corresponding formal parameter. Thus DATE, DATE + 3, and 13 are all suitable as actual parameters for DAY_#, because DAY_# is a formal parameter of type integer.

9.3.2 TWO-WAY (VAR) PARAMETERS

The method of passing parameters illustrated in figure 9-6 allows only one-way passing of parameters; any results computed by the procedure must be assigned to global variables if they are to be referenced outside the procedure. A second type of formal parameter can be specified that allows two-way parameter passing. With this type of parameter, assignments to the formal parameter are assignments to the actual parameter; that is, the procedure returns results by assigning values to a formal parameter. The procedure from figure 9-6 is modified in figure 9-7 to illustrate two-way parameters.

```
TYPE
  days = 0 .. 6,
  total_array = array[days] of integer;

VAR
  first_day : days,
  date : integer,
  tot01 : total_array;

PROCEDURE count_days (     first : days,
                           day_# : integer;
                       VAR total : total_array);
  VAR
    day : days;

  day := (first + day_#) MOD 7;
  total[day] := total[day] + 1;

PROCEND count_days;

date := 193;
first_day := 0;

count_days (2, date, tot01);
count_days (first_day, date + 3, tot01);
count_days (first_day, 13, tot01);
```

Figure 9-7. Two-Way Parameter Passing

The third parameter in procedure COUNT_DAYS (figure 9-7) is a two-way parameter, as indicated by the reserved word VAR preceding the parameter identifier (TOTAL). A semicolon separates its declaration from the one-way parameters that precede it. If additional one-way parameters were added after the declaration of TOTAL, a semicolon would precede their declaration, and VAR would be omitted.

During a procedure's execution, an assignment to a two-way parameter is an assignment to the variable specified as the actual parameter in the procedure call statement. Thus, the assignment to TOTAL[DAY] in procedure COUNT_DAYS is actually an assignment to TOTO1[DAY], because TOTO1 is specified in the procedure call statements in figure 9-7.

The actual parameter specified for a two-way parameter in a procedure call statement must be a variable whose type matches the formal parameter's. It cannot be a constant or an expression. Thus, TOTO1 is an acceptable actual parameter for TOTAL since it is a variable of type TOTAL_ARRAY, the type required for TOTAL by its parameter declaration. CYBIL places an additional requirement on an actual parameter if it is an element of a packed record: such an element must be aligned if it is to be used as an actual parameter.

9.3.3 KEYWORD SPECIFICATION OF ACTUAL PARAMETERS

Correspondence between the actual parameters specified when a procedure is called and the procedure's formal parameters can be achieved in two ways. The first (as described above) is by specifying the actual parameters in the same order as the formal parameters. The second is by assigning the actual parameter to the formal parameter in the procedure call statement as illustrated in figure 9-8.

```
TYPE
  days = 0 .. 6,
  total_array = array[days] of integer;

VAR
  first_day : days,
  date : integer,
  tot01 : total_array;

 PROCEDURE count_days (    first : days,
                           day_# : integer;
                       VAR total : total_array);
   VAR
     day : days;

   day := (first + day_#) MOD 7;
   total[day] := total[day] + 1;
```

```
PROCEND count_days;

date := 193;
first_day := 0;

count_days (2, date, tot01);
count_days (first := first_day, day_# := date + 3, total := tot01);
count_days (total := tot01, first := first_day, day_# := 13);
```

Figure 9-8. Keyword Parameter Specification

The three procedure call statements in figure 9-8 perform identically to those in figure 9-7. The first call is unchanged. The actual parameters in the second and third call statements are specified by keyword, that is, the formal parameter identifier and an assignment operator precede each actual parameter. When specified by keyword, the order of the parameters in a call statement is immaterial. Thus, DAY_# := 13 (in the third call statement of figure 9-8) effectively associates 13 with the second formal parameter declared for procedure COUNT_DAYS, even though it appears as the third actual parameter in the call statement.

If any actual parameters are specified by keyword, then no parameter can be specified positionally. For example,

```
COUNT_DAYS (FIRST := 2, DATE, TOT01)
```

is not a legal CYBIL statement.

9.3.4 DEFAULT PARAMETER VALUES

CYBIL provides an alternate mechanism for assigning a value to a formal one-way (nonVAR) parameter when the procedure is called whereby no actual parameter value need be specified. Rather, a default value is specified with the parameter's declaration in the procedure header; if no corresponding actual parameter is specified when the procedure is called, the formal parameter takes on the default value.

A default value can be specified for only a one-way (nonVAR) parameter, and must be a constant (not a constant expression). It is placed after the formal parameter's type identifier in the procedure header, separated by := , as shown in figure 9-9.

```
TYPE
  days = 0 .. 6,
  total_array = array[days] of integer;

VAR
  first_day : days,
  date : integer,
  tot01 : total_array;

PROCEDURE count_days (      first : days := 0,
                            day_# : integer;
                        VAR total : total_array);
  VAR
    day : days;

  day := (first + day_#) MOD 7;
  total[day] := total[day] + 1;

PROCEND count_days;

date := 193;

count_days (2, date, tot01);
count_days ( , date + 3, tot01);
count_days (total := tot01, day_# := 13);
```

Figure 9-9. Default Parameter Value

The program segment in figure 9-9 performs identically to the one shown in figure 9-8. (Note that the assignment statement FIRST := 0 is deleted.) Since a value (2) is explicitly specified for FIRST in the first procedure call, the default value is ignored. The parameter is omitted from the second procedure call, as indicated by the absence of a value preceding the first comma; FIRST takes on a value of 0 by default. Since the parameters in the third procedure call are specified by keyword, the lack of any value equated to FIRST results in the default value being used. Had no default value been specified for FIRST in the procedure header, the second and third procedure calls in figure 9-9 would be in error. An actual parameter must always be specified for a two-way (VAR) parameter.

9.4 XDCL AND XREF ATTRIBUTES

XDCL and XREF attributes were discussed with respect to variables in section 3. These attributes have equivalent functions for procedures. They are needed only when declaring a procedure in one module that is to be called from another module.

Figure 9-10 illustrates this kind of referencing mechanism.

```
MODULE first;                        MODULE second;

  PROCEDURE [XREF] compute;            PROCEDURE [XDCL] compute;
  PROCEDURE [XDCL] main;                 .
    .                                    .
    .                                  PROCEND compute;
  compute;
    .                                MODEND second;
    .
  PROCEND main;

MODEND first;
```

Figure 9-10. XDCL and XREF Procedures

In figure 9-10, modules FIRST and SECOND would be compiled separately and then loaded. Procedure COMPUTE is declared in module SECOND and (in module SECOND) is given the attribute XDCL. This means that the procedure is declared in this module and can be referenced from another module.

In module FIRST only the line "PROCEDURE [XREF] compute;" appears to identify COMPUTE. This indicates that COMPUTE identifies a procedure. The XREF attribute indicates that the procedure is declared in a different module.

If procedure COMPUTE had parameters, the parameter list would be specified in each procedure declaration in each module. The parameter specifications must agree in number, order, and type; the identifiers of corresponding parameters can, however, be different.

Procedures with the XDCL attribute cannot be nested. That is, they must be declared at the outermost level.

9.5 INITIATING PROGRAM EXECUTION

As described so far, a procedure is executed only when it is called by a CYBIL procedure call statement. If this were always true, no procedure would ever be executed, since no mechanism has been discussed that causes the first procedure to be called by the operating system. While operating system commands (control statements and so forth) for loading and executing a compiled CYBIL program are not presented here, the mechanism is described whereby the first procedure to be executed is identified within a group of modules that are loaded and executed as a single program.

An alternate form of the procedure header identifies a procedure as the first one to be executed. It begins with the reserved word PROGRAM in place of PROCEDURE and cannot include attributes (XREF or XDCL). Furthermore, its parameter list (if present) is subject to restraints imposed by the operating system command language (not described here). Figure 9-10 contains an example of this type of procedure declaration.

```
PROGRAM main (optional parameter list);
  .
  .
  { Procedure body }
  .
  .
PROCEND main
```

Figure 9-10. PROGRAM Procedure Declaration

A PROGRAM procedure can still be called via a procedure call statement like any other procedure. The reserved word PROGRAM serves only to identify the procedure as the first to be executed. As such, only one PROGRAM procedure can exist among any group of compilation units loaded and executed together.

9.6 RETURN STATEMENT

When a procedure completes execution, the execution of the procedure from which it was called resumes. This can happen in two ways: by the execution of the last statement in the procedure's statement list, or by execution of a RETURN statement. A RETURN statement, when executed, causes control to return immediately to the calling procedure. The statement consists simply of the reserved word RETURN.

10.0 ADAPTABLE TYPES

The types discussd in the preceding sections (scalar types, array, string, record, and set) share a characteristic that, until now, need not be mentioned: they are *fixed* types. That is, the space required to store a variable of any of these types is constant and can be determined before program execution begins. For example, a character variable requires an amount of space that is fixed by the implementation; an array whose index is [1 .. 5] requires space for five variables of its component type, and so on for other fixed types.

Certain programming situations, however, prevent the use of fixed types or make their use awkward and inefficient. (These situations include procedures that process paramaters of indeterminate size and procedures that involve dynamic storage allocation techniques.) Such situations call for a flexible type definition that can be fixed during program execution: adaptable types.

The adaptable types described in this section include adaptable arrays, adaptable strings, and adaptable records. These adaptable types may be used in only two ways: as formal parameters in a procedure, and as components allocated by storage management statements. In each of these uses, the size of the adaptable type must be fixed during program execution. In the case of formal parameters of procedures, the actual parameter fixes the size of the adaptable type. The use of adaptable types in storage management statements requires an understanding of pointers (covered in section 11) and other storage management features, and is covered in section 12, Storage Management.

10.1 ADAPTABLE TYPE DECLARATION

Adaptable types are declared by specifying an asterisk in place of the the type's size. The asterisk replaces the array bounds or string length in an array or string declaration. Examples of adaptable arrays and strings are given in figure 10-1.

```
TYPE
  adaptarray = array[*] OF integer,
  adaptstring = string (*),
  fixedarray = array[7] OF integer,
  adapt2d = array[*] OF fixedarray;
```

Figure 10-1. Adaptable Arrays and Strings

The type ADAPTARRAY in figure 10-1 is an adaptable array of integers. The asterisk indicates that the index is type integer (by convention), but the array's size (dimension) is unspecified. The type ADAPTSTRING is an adaptable string of characters. The length of the string is adaptable and must be fixed during program execution, as indicated by the asterisk.

The last two lines of figure 10-1 illustrate a two-dimensional adaptable array. The outermost array dimension of ADAPT2D is adaptable; the inner dimension is fixed. This construction obeys the CYBIL rule that permits only a single adaptable dimension for an array, and requires that dimension to be the first (outermost). Several invalid type definitions that violate this rule are illustrated in figure 10-2 (assuming the types defined in figure 10-1 remain in effect).

```
TYPE
  wrong_2d_array = array[7] OF adaptarray,
  wrong_string_array = array[7] OF adaptstring,
  wrong_again = array[*] OF adaptarray,
  still_wrong = array[*] OF adaptstring;
```

Figure 10-2. Invalid Adaptable Arrays

An adaptable record is a record containing zero or more fixed fields followed by one and only one adaptable field. The example in figure 10-3 illustrates the declaration of adaptable records.

```
TYPE
  model_descriptor = record
    number : string (8),
    name : string (*),
  recend,
  origin_codes = (do, me, co, ac),
  assembly = record
    origin : origin_codes,
    date : string (8),
    model : model_descriptor,
  recend;

VAR
  part : model_descriptor,
  product : assembly;
```

Figure 10-3. Adaptable Records

10.2 ADAPTABLE FORMAL PARAMETERS

Adaptable formal parameters of a procedure allow the procedure to adapt to the size of the actual parameter at execution time.

Consider, for example, a procedure that sums the squares of all the elements of an array (which is passed as an actual parameter) and returns the sum via a VAR parameter. If this procedure is to be a general one that operates on an array of any length, the array must be adaptable and the procedure must have available the number of elements in the array. During execution the formal array parameter adapts to the size of the actual parameter passed; size determining functions (described shortly) provide the values of the array's bounds.

The procedure header for such a procedure might appear as follows:

```
PROCEDURE sum_squares (     data : array[*] OF integer;
                        VAR sum : integer);
```

10.3 SIZE_DETERMINING_FUNCTIONS

The typical uses of adaptable arrays and strings require an ability to determine the bounds of the adaptable array and the length of the adaptable string during execution.

10.3.1 LOWERBOUND and UPPERBOUND

The functions LOWERBOUND and UPPERBOUND determine the bounds of an adaptable formal array parameter. The general form of these functions is shown in figure 10-4.

LOWERBOUND (array identifier)

UPPERBOUND (array identifier)

Figure 10-4. LOWERBOUND and UPPERBOUND General Form

The array identifier (see figure 10-4) specifies the formal adaptable array parameter whose upper or lower bound is desired. Since the bounds of an adaptable array are implicitly type integer, these functions return a value of type integer when used upon an adaptable array.

10.3.2 STRLENGTH

The effective use of an adaptable string within a procedure requires a method of obtaining the actual string length. The STRLENGTH standard function provides this capability (see figure 10-5).

STRLENGTH (string variable identifier)

Figure 10-5. STRLENGTH Standard Function

The function parameter identifies the string variable (it may also identify a string type). The function returns an integer value indicating the length of the string.

10.4 EXAMPLES

An implementation of the array squaring procedure is shown in figure 10-6.

```
TYPE
  adaptarray = array[*] of integer;

PROCEDURE squarearray (VAR data : adaptarray);

  VAR
    index : integer;

  FOR index := LOWERBOUND (data) TO
               UPPERBOUND (data) DO
    data[index] := data[index] * data[index];
  FOREND;
PROCEND squarearray;
```

Figure 10-6. Adaptable Array Example

The procedure SQUAREARRAY (in figure 10-6) has an adaptable array formal parameter, DATA. The FOR statement uses the functions LOWERBOUND and UPPERBOUND to determine the bounds of DATA. The FOR statement then iterates the proper number of times (with the proper values of INDEX) to compute the square of each value in the array passed as an actual parameter.

SQUAREARRAY can be called with any array of integers indexed by an integer subrange. Some examples of calls to the procedure SQUAREARRAY are shown in figure 10-7.

```
VAR
  x : array[-10 .. 10] of integer,
  y : array[1 .. 100] of integer;
  .
  .
squarearray (x);
squarearray (y);
```

Figure 10-7. Actual Array Parameters

In figure 10-7, the procedure SQUAREARRAY from figure 10-6 is called. The array X is passed in the first call; array Y is passed in the second call. SQUAREARRAY adapts to the different sizes of arrays because the formal parameter (declared in procedure SQUAREARRAY) is an adaptable array.

The following example illustrates the use of adaptable strings as formal parameters. The example depicts a procedure that counts the number of blanks in the actual parameter (a string variable). The procedure returns the number of blanks found via a two-way parameter, NBLANKS.

```
TYPE
  astring = string (*);

PROCEDURE blank_count (     in_str : astring;
                       VAR nblanks : integer);

  VAR
    position : integer;

  nblanks := 0;
  FOR position := 1 TO STRLENGTH (in_str) DO
    IF in_str (position) = ' ' THEN
      nblanks := nblanks + 1;
    IFEND;
  FOREND;

PROCEND blank_count;
```

Figure 10-8. Adaptable String Example

In figure 10-8, the formal parameter IN_STR is an adaptable string. NBLANKS is a two-way formal parameter to enable the return of the number of blanks.

BLANK_COUNT begins by clearing the blanks counter ("NBLANKS := 0;"). The FOR statement checks each character in the string, incrementing NBLANKS with each occurrence of a space. STRLENGTH determines during execution the length of the actual string and controls the number of iterations performed by the FOR statement.

NBLANKS is cleared whenever BLANK_COUNT executes. A static variable initialized to zero would not be acceptable here because it would be set to zero at the procedure's first execution and would thereafter serve as a running accumulator, returning the grand total of the number of blanks in all strings processed by BLANK_COUNT.