Table of Contents

LANGUAGE SPECIFICATION

for the

CYBER IMPLEMENTATION LANGUAGE

(CYBIL)

Written By: _____
H.A.Wohlwend

Approved By: _____
L.L.Bumgarner

_____
E.H.Michehl, AD&C

DISCLAIMER:

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

REVISION DEFINITION SHEET

REV	DATE	DESCRIPTION
1	10/07/77	Original.
2	12/19/77	Updated to reflect comments received through the DCS review.
3	06/27/78	Updated to reflect V2.0 of the language definition.
4	10/16/78	Updated to reflect comments received through the DCS review.
5	12/07/79	Updated to reflect approved DAP's and miscellaneous clarifications.
6	06/01/81	Updated to reflect approved DAP's and miscellaneous clarifications.

1.0 INTRODUCTION

1.0 INTRODUCTION

The CYBIL language is intended to be used as the system implementation language for Control Data Corporation. This document provides the definition for the CYBIL language. This specification was developed from Rev. 5 of this specification and from DAP's S3470, S3471, S3484, S3485, S3527, S3654, S3691, S3873 and S3899.

These DAP's have Implementation Language Design Team and DCS review cycle approval.

2.0 LANGUAGE OVERVIEW

2.0 LANGUAGE_OVERVIEW

A CYBIL program consists of *statements*, which define actions involving programmatic elements, and *declarations*, which define such elements.

The definable elements include *variables* and *procedures*, all having the characteristics that are conventionally associated with their names. Declarations of instances of variables are spelled out in terms of an *identifier* for the element and a *type* description, which defines the operational aspects of the element and, in many cases, indicates a notation for referencing. In the case of a variable declaration, the type defines the set of values that may be assumed by the variable. Types may be directly described in such declarations, or they may be referenced by a type identifier, which in turn must be defined by an explicit type declaration. A small set of pre-defined types are provided, together with notations for defining new types in terms of existing ones.

In general, an element may not enter into operations outside the domain indicated by its type, and most dyadic operations are restricted to elements of equivalent types (e.g., a character may not be added to an integer). Since the requirements for type equivalence are severe, these operational constraints are strict. Departures from them must be explicitly spelled-out in terms of *conversion functions*.

The *basic types* include the pre-defined *integer*, *char*, and *boolean* types, all having their conventional connotations, value sets, and operational domains. These are *scalar types*, which define well-ordered sets of values. A scalar type may also be defined as an *ordinal type* by enumerating the identifiers which stand for its ordinal values, or as a *subrange* of another scalar type by specifying the smallest and largest values of the subrange. Also included in the basic types are the floating point types: *real* and *longreal* types. *Pointer types* are included in the basic types. They represent location values, and other descriptive information, that can be used to reference instances of variables and other CYBIL elements. Pointers are bound to specific types, and pointer variables may assume, as values, only pointers to elements of those types. *Cell types* are also included in the basic types. Cells represent the smallest addressable memory unit supported by an implementation.

Structured types represent collections of components, and are

2.0 LANGUAGE OVERVIEW

defined by describing their component types and indicating a so-called *structuring method*. These differ in the accessing discipline and notation used to select individual components. Four structuring methods are available: *set* structure, *string* structure, *array* structure, and *record* structure.

A *set* type represents all subsets of values of some scalar type.

A *string type* of length *n* represents all ordered *n*-tuples of values of character type. An ordered *k*-tuple of these values ($1 \leq k \leq n$) is called a *substring*. Notation for accessing substrings is provided.

An *array type* represents a structure consisting of components of the same type. Each component is selected by an *array selector* consisting of an ordered set of *n* index values whose types are indicated in the array definition.

A *record type* represents a structure consisting of a fixed number of components called fields, which may be of different types and which must be identified by *field selectors*. In order that the type of a selected field be evident from the program text (without executing the program) a field selector is not a computable value, but instead is an identifier uniquely denoting the component to be selected. These component identifiers are declared in the record type definition.

A *variant record type* may be specified as consisting of several *variants*. This implies that different variables, although said to be of the same type, may assume structures which differ in a certain manner. The difference may consist of a different number and different types of components. The variant which is assumed by the current value of a record variable is indicated by a component field which is common to all variants and is called the *tag* field.

Array and record types may have associated *packing attributes* which can be used to specify component space-time trade-offs. Access time for specific components of packed (space-compressed) structures can be shortened by declaring them to be aligned. *Aligned* also provides a method of specifying specific hardware boundaries.

Storage types represent structures to which other variables may be added, referenced, and deleted under explicit program control. There are two storage types, each with its own management and access characteristics. *Sequence types* and *heap types* represent storage structures whose components may be of diverse type. Components of *sequences* are managed through the operations of resetting to the first component and moving to the *next* component and are accessed through pointers constructed as by-products of the *next* operator. Space for components of *heap storages* must be explicitly managed by

the operation of allocate and free; the components are accessed through pointers constructed as by-products of the allocate operation.

Adaptable types are array, record, string, sequence and heap types defined in terms of one indefinite bound. They may be used as formal parameters of procedures -- in which case the bounds of the actual parameters are assumed; or they may be used to define pointers to structures which are meant to be explicitly fixed during execution of the program.

Denotations for explicit values of the basic and structured types consist of constants and constant expressions, which denote constant values of the basic and string types; and value constructors which are used to denote instances of values of set, array, and record types. The boolean constants (false,true) are pre-defined. New constants can be introduced by constant declarations, which associate an identifier with a constant expression.

Set value constructors, which include set type information, may be used freely in set expressions. Indefinite set value constructors can be used only in initialization of variables where their type is explicitly indicated by the context in which they occur.

Variables can be declared with initialization specifications and with certain attributes. Initialization expressions are evaluated when storage for the static variable is allocated, and the resultant values are then assigned to the variable. The attributes include access attributes - which specify the purposes for which the variable may be accessed; storage attributes - which specify when storage for the variable is to be allocated and when it is to be freed; and scope attributes -which specify the program span over which the declaration is to hold (the scope of the declaration). Unless otherwise specified, the scope of a declaration is the block containing the declaration, including all contained sub-blocks except for those which contain a re-declaration of the identifier.

Blocks are portions of programs which are grouped together as procedures or functions, and used to define scope and to provide shielding of identifiers. Procedures or functions have identifiers associated with them, so that the identified portions of the program can be activated on demand by statements of the language.

A procedure is declared in terms of its identifier, the associated program, a set of attributes, and a list of formal parameters. Formal parameters provide a mechanism for the binding of references to the procedure with a set of values and variables - the actual parameters - at the point of activation.

A function returns a value of a specified type. These

return-types are restricted to the basic types, and are specified in the function declaration.

In addition to their other programmatic aspects, blocks provide partial mechanisms for the shielding and sharing of variables and portions of programs. Modules (together with scope attributes) provide a mechanism for the shielding and sharing of declarations. Modules are primarily designed to permit program packaging at the "source" language level.

Statements define actions to be performed.

Structured statements are constructs composed of statement lists: begin statements provide for execution of a list of statements; while, for and repeat statements control repetitive execution of a single statement list.

Control statements cause the creation or destruction of execution environments. They provide for the activation of procedures, and for general changes in the flow of control. If statements provide for the conditional execution of one of a set of statement lists.

Storage management statements provide mechanisms for allocating new local variables, moving forward and backward over components of sequences, and allocating and freeing variables in heaps.

A set of pre-defined procedures and functions exists which can be used for storage management, scalar conversions, etc.

Finally, assignment statements cause variables to assume new values.

Compile-time facilities, that are essentially extra-linguistic in nature, are used to control the compilation process and construct the program to be compiled; these include compile-time variable declarations, and compile-time statements.

3.0 METALANGUAGE AND BASIC CONSTRUCTS

3.0 METALANGUAGE AND BASIC CONSTRUCTS

3.1 METALANGUAGE

In this specification, syntactic constructs are denoted by English words enclosed between angle brackets < and >. These words also describe the nature or meaning of the construct, and are used in the accompanying description of semantics.

Constructs not enclosed in angle brackets stand for themselves.

The symbol ::= is used to mean "is defined as", and the vertical bar | is used to signal an alternative definition.

An optional syntactic unit (zero or one occurrences) is designated by square brackets [and].

Indefinite repetition (zero or more occurrences) is designated by braces { and }.

Examples:

The definition:

```
<field> ::= <fixed field>
          | <variant field>
```

is read: " a field is either a fixed field or a variant field."

The definition:

```
<fixed field> ::=
          <field selectors> : <type>
```

is read: "a fixed field consists of field selectors, followed by a colon, followed by a type."

The definition:

```
<field selectors> ::=
          <field selector>{,<field selector>}
```

is read: "field selectors consist of a field selector, followed by

3.0 METALANGUAGE AND BASIC CONSTRUCTS
3.1 METALANGUAGE

zero or more comma separated field selectors."

The angle brackets, square brackets, and braces are also elements of the language, and therefore are used in syntactic constructs. Such syntactic occurrences of these symbols will be underscored when necessary.

Example:

The definition:

<attributes> ::= [<attribute >{,<attribute>}]

is read as, "attributes consist of an attribute followed by zero or more comma-separated attributes, the entire set of attributes being enclosed in square brackets."

Words reserved for specific purposes in the language will always be underscored.

Example:

The definition:

```
<array spec> ::=
       array [<index>] of <component type>
```

is read as, "an array spec is composed of the word 'array' followed by an index enclosed in square brackets, followed by the word 'of' followed by a component type."

Appendix A of this specification contains a sorted alphabetic list of all constructs in the syntax with their definitions.

3.2 LEXICAL CONSTRUCTS

The lexical units of the language - identifiers, basic symbols, and constants - are constructed from one or more (juxtaposed) elements of the alphabet.

3.2.1 ALPHABET

The alphabet consists of tokens from a subset of the 256-valued ASCII character set: those for which graphic denotations are defined.

3.0 METALANGUAGE AND BASIC CONSTRUCTS
3.2.1 ALPHABET

```
<ascii character> ::= <alphabet>
                    ¦<unprintable>
                    ¦<string delimiter>

<alphabet> ::= <letter>
             ¦<digit>
             ¦<special mark>
             ¦<blanks>
             ¦<unused mark>

<letter> ::= 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
           ¦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

<digit> ::= 0¦1¦2¦3¦4¦5¦6¦7¦8¦9

<string delimiter> ::= '

<special mark> ::= +¦-¦*¦/¦.¦;¦:¦,
                 ¦#¦$¦_¦@¦?¦(¦)¦=¦<¦>¦[¦]¦^¦{¦}

<blanks> ::=

<unused mark> ::= &¦%¦¦¦~¦`¦\¦!¦"
```

3.2.2 IDENTIFIERS

Identifiers serve to denote constants, variables, procedures, and other programmatic elements of the language.

```
<identifier> ::= <letter>{<follower>}

<follower> ::= <letter>¦<digit>
             ¦_¦#¦$¦@
```

Identifiers are restricted to a maximum of 31 characters, and identifiers that differ only by case shifts of component letters are considered to be identical. Identifiers must begin with a letter and may not contain embedded blanks. An exception is made to this rule for the system dependent functions and procedures which begin with the # character.

Examples_of_Valid_Identifiers:

```
x2    Henry    Job#    A_wordy_Identifier
```

Examples_of_Invalid_Identifiers:

```
1st_character_must_be_a_letter
number_of_characters_must_not_exceed_thirtyone
```

3.2.3 BASIC SYMBOLS

Selected identifiers, special marks and digraphs of special marks are reserved for specific purposes in the language; e.g., as operators, separators, delimiters. These so-called "basic symbols" will be introduced as they arise in the sequel.

Identifiers reserved for use as basic symbols will be shown as underscored, lower-case words.

3.2.4 CONSTANTS

Constants are lexical constructs used to denote values of some of the elementary data types. Their spellings, and the data types for which constant denotations can be given, are described in Section 5.1.1.

3.2.5 CONVENTIONS FOR BLANKS

Identifiers, reserved words, and constants must not abut each other, and must not contain embedded blanks, except string constants. Identifiers, reserved words, string terms and non-string constants must be contained on one input line. Basic symbols constructed as digraphs may not contain embedded blanks. Otherwise, blanks may be employed freely, and have no effect outside of character constants and string constants - where they represent themselves.

3.2.6 COMMENTS

Commentary strings may be used anywhere that blanks may be used except within character and string constants.

```
<commentary string> ::= [{<comment character>}
                         <comment terminator>

<comment terminator> ::= } ! <end of line>

<comment character> ::= <any ASCII character except
                         a closing brace or end of line>
```

4.0 CYBIL TYPES

4.0 CYBIL_TYPES

CYBIL types are used to define operational domains and characteristics of variables (which take on values) and other programmatic elements. CYBIL elements fall into two broad classes of types.

```
<type> ::= <fixed type>
         ¦<fixable type>
         ¦<procedure type>

<fixable type> ::= <adaptable type>
                 ¦<bound variant record type>
```

Fixed types are used to define sets of values that can be assumed by CYBIL variables, their operational domain and, in many cases, a notation for referencing such values.

Fixable types are associated with data types whose precise attributes are meant to be explicitly "fixed" during execution of the program. Variables of a fixable type must be referenced in an indirect manner; they may be referenced through the use of a pointer or as a formal parameter of a procedure.

4.1 TYPE_DECLARATIONS

CYBIL provides a small set of pre-defined types, reserved identifiers for these, and notation for defining new types in terms of existing ones.

Type declarations are used to introduce new types, and identifiers for the newly declared types.

4.0 CYBIL TYPES
4.1 TYPE DECLARATIONS

<type declaration> ::=
type <type spec>{, <type spec>}

<type spec> ::= <identifier> = <type>

Type declarations can be used for purposes of brevity, clarity, and accuracy. Once declared, a type may be referred to elsewhere by its declared type identifier. The identifier can have mnemonic value and errors associated with repeated spelling-out of type specifications are reduced.

4.2 TYPE MATCHING

In general, operations involving elements of non-equivalent types are not allowed, and one type may not be used where another type is expected. Relaxations to these rules are sometimes permitted, and will be stated as they arise.

4.2.1 TYPE EQUIVALENCE

Two equivalent types can be expressed differently. For example: a declared type identifier and the type it denotes have different spellings; different expressions for sizes of arrays and other collections of elements can yield the same value; formal parameter identifiers are not part of procedure types.

Rules for determining type equivalence are called-out in the following sections on types.

4.2.2 POTENTIAL EQUIVALENCE, INSTANTANEOUS TYPES

Adaptable types and bound variant record types actually define classes of related types. References to variables of such type are meant to be explicitly fixed to a so-called instantaneous type during the execution of the program. Such types are said to be potentially-equivalent to any of the types to which they can be fixed. Since the determination of that type can be made only during program execution, references to variables of such types are permitted wherever a reference to one of the instantaneous types is valid. No compile-time error messages will be issued; however, each implementation is required to carry out the required execution-time checks for type-matching when selected by the programmer, and to report violations (see Compile-Time Facilitities, Run-Time Checking Toggles).

4.3 FIXED_TYPES

Data types are used to define sets of values that may be assumed by variables.

Fixed types consist of:

A) Basic types, which take on simple values.

B) Structured types, which define collections of components.

C) Storage types, which are used as repositories for collections of components of various types.

```
<fixed type> ::= <basic type>¦<structured type>¦<storage type>
```

4.3.1 BASIC TYPES

Basic types define components that take on simple values.

```
<basic type> ::= <scalar type>
                 ¦<floating point type>
                 ¦<pointer type>
                 ¦<cell type>
```

4.3.1.1 Scalar_Types

Scalar types define well-ordered sets of values for which the following functions are defined:

succ the succeeding value in the set;
pred the preceding value in the set.

```
<scalar type> ::= <integer type>
                  ¦<character type>
                  ¦<ordinal type>
                  ¦<boolean type>
                  ¦<subrange type>
```

4.3.1.1.1 INTEGER TYPE

```
<integer type> ::= integer¦<integer type identifier>

<integer type identifier> ::= <identifier>
```

4.0 CYBIL TYPES
4.3.1.1.1 INTEGER TYPE

Integer type represents an implementation-dependent subset of the integers, and is equivalent to the subrange defined by

-n1 .. n2

where n1 and n2 denote implementation-dependent integers. In general, if transportation of programs is planned across implementations, the explicit use of integer types should be avoided in favor of subrange types.

Permissible operations: assignment, set membership test, all relational operators, addition, subtraction, multiplication, quotient, remainder and applicable standard procedures and functions.

4.3.1.1.2 CHARACTER TYPE

<character type> ::= char|<character type identifier>

<character type identifier> ::= <identifier>

Character type defines the set of 256 values of the ASCII character set, and is equivalent to the subrange defined by

chr(0) .. chr(255)

where "chr" denotes the mapping function from integer type, onto character type. Characters may be assigned & compared to strings.

Permissible operations: assignment, set membership test, all relational operators, standard procedures and functions.

4.3.1.1.3 ORDINAL TYPE

```
<ordinal type> ::=
              (<ordinal constant identifier list>)
             |<ordinal type identifier>

<ordinal constant identifier list> ::=
     <ordinal constant identifier>
        ,<ordinal constant identifier>
           {,<ordinal constant identifier>}

<ordinal constant identifier> ::= <identifier>
<ordinal type identifier> ::= <identifier>
```

An ordinal type defines an ordered set of values by enumeration, in the ordinal list, of the identifiers which denote the values.

4.0 CYBIL TYPES
4.3.1.1.3 ORDINAL TYPE

Each of the identifiers (at least two) in the ordinal list is thereby declared as a constant of the particular ordinal type.

Two ordinal types are equivalent if they are defined in terms of the same ordinal type identifier.

Permissible operations: assignment, set membership test, all relational operators, standard procedures and functions.

Example: The constants of the ordinal type "primary color" declared by

```
type primary_color = (red, green, blue)
```

are denoted by "red", "green", and "blue", and the following relations hold:

```
red < green
red < blue
green < blue
```

A mapping from ordinals onto non-negative integers is provided by the ord function. For the constants of the example, the following relations hold:

```
ord (red) = 0
ord (green) = 1
ord (blue) = 2
```

The ordinal type declaration

```
type primary_color = (red, green, blue),
          hot_color = (red, orange, yellow)
```

would be in error because of the dual definition of the identifier "red" as a constant of two different ordinal types.

4.3.1.1.4 BOOLEAN TYPE

```
<boolean type> ::= boolean
                 |<boolean type identifier>

<boolean type identifier> ::= <identifier>
```

4.0 CYBIL TYPES
4.3.1.1.4 BOOLEAN TYPE

Boolean type represents the ordered set of "truth values", whose constant denotations are false and true, and is conceptually equivalent to the ordinal type specified by:

(false,true), except that Boolean operations are permitted on Boolean types.

Permissible_operations: assignment, set membership test, all relational operators (false < true), the Boolean operations of sum, product, difference, negation and standard procedures and functions.

4.3.1.1.5 SUBRANGE TYPE

```
<subrange type> ::= <subrange type identifier>
                  |<lower>..<upper>

<lower> ::= <constant scalar expression>
<upper> ::= <constant scalar expression>

<subrange type identifier> ::= <identifier>
```

The lower bound must not be greater than the upper bound and both must be of equivalent scalar types. Two subrange types are equivalent if they have identical upper and lower bounds. An improper subrange type (i.e., one that completely spans its 'parent' range) is equivalent to its parent type.

Values of a subrange and values of its parent range (or values of other subranges of its parent type) may enter jointly into dyadic operations defined for the parent type, and into assignment operations; execution time checks on the validity of such assignments may be specified (see Run-Time Checking Toggles).

Permissible_operations: same as for the parent type.

Example:

```
type  non_negative integer = 0..32767,
      letter = 'A'..'Z',
      color = (red, orange, yellow, green, blue),
      hotcolor = red..yellow,
      range = -10..10 ;
```

4.3.1.2 Floating_Point_Type

```
<floating point type> ::= <real type> | <longreal type>
```

The floating point types define values that approximate the real

numbers and which are to be represented in a machine-dependent form of scientific notation. The real and longreal types are intended to have the same representation as FORTRAN REAL and DOUBLE PRECISION, respectively.

4.3.1.2.1 REAL TYPE

<real type> ::= real |<real type identifier>

<real type identifier> ::= <identifier>

The range and precision of the real type are implementation-dependent. Conversion functions between real, longreal and integer type are provided (cf. Standard Functions, 11.2).

Permissible operations: assignment, all relation operators, addition, subtraction, multiplication, division, and applicable standard procedures and functions.

4.3.1.2.2 LONGREAL TYPE

<longreal type> ::= longreal |<longreal type identifier>

<longreal type identifier> ::= <identifier>

The range and precision of the longreal type are implementation-dependent. Conversion functions between real, longreal and integer type are provided (cf. Standard Functions, 11.2).

Permissible operations: assignment, all relation operators, addition, subtraction, multiplication, division, and applicable standard procedures and functions.

4.3.1.3 Pointer Type

Pointer types represent location values, and other descriptive information, that can be used to reference instances of CYBIL objects indirectly.

Permissible operations: assignment, comparison for equality and inequality only, and standard procedures and functions.

Pointer types are introduced by an up arrow, followed by a CYBIL type to which the pointers are bound; pointer variables may assume, as values, only pointers to that type. The only exception to this is pointer to cell.

4.0 CYBIL TYPES
4.3.1.3 Pointer Type

```
<pointer type> ::= <fixed pointer>
                  ¦<fixable pointer>
                  ¦<pointer to procedure>
                  ¦<pointer type identifier>

<fixed pointer> ::= ^<fixed type>

<fixable pointer> ::= <adaptable pointer>
                     ¦<bound variant pointer>

<adaptable pointer> ::= ^<adaptable type>

<bound variant pointer> ::= ^<bound variant record type>

<pointer to procedure> ::= ^<procedure type>

<pointer type identifier> ::= <identifier>
```

Adaptable pointers provide the sole mechanism for accessing objects of adaptable type, other than through formal parameters of procedures. In particular, adaptable pointers and bound variant pointers are used to access instances of adaptable variables and bound variant records whose type has been 'fixed' by an allocate or a next statement.

Pointers are equivalent if they are defined in terms of equivalent types. A pointer to a fixed type may be assigned and compared to an adaptable pointer or bound variant record pointer if the adaptable type is potentially equivalent to the fixed type.

See Section 10.2, Assignment Statements, for rules governing pointer assignment.

4.3.1.3.1 POINTER TO CELL

```
<pointer to cell> ::= ^cell
```

A pointer to cell is a pointer type.

Permissible Operations: as for pointers; in addition, pointers to cell may be assigned to any pointer to fixed or bound variant type. Such an assignment must not result in a pointer to fixed or bound variant type having as its value a pointer to a variable that is not of cell type and whose type is not equivalent to that to which the target of the assignment is bound. Pointer to cell may be the target of assignment of any pointer to fixed or bound variant type.

4.3.1.4 Cell_Type

<cell type> ::= cell

A cell type is a basic type that represents the smallest storage site that is directly addressable by a pointer.

Permissible_Operations: assignment, comparison for equality and inequality only, and applicable standard functions.

4.3.2 STRUCTURED TYPES

Structured types represent collections of components, and are defined by describing their component types and indicating a so-called structuring method. These differ in the accessing discipline and notation used to select individual components. Four structuring methods are available: set structure, string structure, array structure, and record structure. Each will be described in the sequel.

```
<structured type> ::= <set type>
                     !<aggregate type>

<aggregate type> ::= <string type>
                    !<array type>
                    !<record type>
```

4.3.2.1 Set_Type

```
<set type> ::= set of <base type>
              !<set type identifier>

<base type> ::= <scalar type>

<set type identifier> ::= <scalar identifier>

<scalar identifier> ::= <identifier>
```

A set type represents the set of all subsets of values of the base type. The number of elements defined by the base type must be constrained (consider, e.g., set of integer). The number of elements will be implementation dependent, but no less than 256 (to accommodate set of char).

Set types are equivalent if they have equivalent base types.

Permissible____operations: assignment, intersection, union,

difference, symmetric difference, negation, inclusion, identity, membership.

Example: The set, akcess, declared by

type akcess = set of (no_read, no_write, no_execute)

represents the set of the following subsets of values of its ordinal base type:

```
$akcess [ ] {the empty set}
$akcess [no_read]
$akcess [no_write]
$akcess [no_execute]
$akcess [no_read, no_write]
$akcess [no_read, no_execute]
$akcess [no_write, no_execute]
$akcess [no_read, no_write, no_execute] {the full set}
```

where the notation "$akcess [...]" denotes a value constructor for the set type, akcess. Note that succ and pred are not defined for set types. The values of a set variable are only partially ordered by set inclusion. $akcess [no_read] and $akcess [no_write] satisfy no order relation except inequality.

4.3.2.2 String_Type

A string type represents ordered n-tuples of values of character type.

```
<string type> ::= <fixed string>
                | <string type identifier>

<fixed string> ::= string (<length>)

<length> ::= <positive integer constant expression>

<string type identifier> ::= <identifier>
```

A fixed string of length n represents all ordered n-tuples of values of character type. The length must be a positive integer constant expression in the range 1 to 65535.

An ordered k-tuple of the values of a string (1 <= k <= n) is called a substring. Notation for accessing substrings is provided.

Two string types are equivalent if they have the same length.

Strings of different length may be assigned and compared. The shorter is blank-filled on the right for comparisons and for assignments to longer strings; truncation on the right is applied for assignments to shorter strings. Characters may be compared and assigned to strings of any length, and are treated as strings of length one in such cases. Substrings of length one are treated as characters in several specific instances -- see Substring References as Character References.

Permissible_operations: assignment, comparison (all six relational operators), and standard procedures and functions.

4.3.2.3 Array_Type

An array type represents a structure consisting of components of the same type. Each component is selected by an array selector consisting of an ordered set of n index values whose scalar type is indicated by the indices in the definition.

```
<array type> ::= [packed]<array type identifier>
                 |[packed]<array spec>

<array type identifier> ::= <identifier>

<array spec> ::=
       array [<index>] of <component type>

<index> ::= <scalar type>
   |<constant scalar expression>
           ..<constant scalar expression>

<component type> ::= <fixed type>
```

Two array types are equivalent if they have the same packing, have equivalent component types, and indexes are of equivalent type.

Permissible_operations: assignment and applicable standard procedures and functions.

4.3.2.3.1 PACKED ARRAYS

Packing attributes are used to specify storage space versus access time tradeoffs for array components. Components of a packed array will be mapped onto storage so as to conserve storage space at the possible expense of access time. The array itself (the collection of components) is always mapped onto an addressable memory location.

4.3.2.3.2 EXAMPLES OF ARRAY TYPE

```
type  hotness = array [color] of non_negative_integer,
      token_code = array [char] of token_class,
      array1 = array [100..200] of 100..300,

      i1 = 1..100,
      i2 = 100..200,
      s1 = 100..300,

      array2 = array [i1] of array1,
      array2b = array [i1] of array [i2] of s1;
```

The array types 'array2' and 'array2b' are alternate ways of defining an array of arrays.

4.3.2.4 Record_Type

A record type represents a structure consisting of a fixed number of components called fields. Fields are defined in terms of their types and associated field selectors, which are identifiers uniquely denoting that field among all other fields of the record.

Permissible_operations: assignment, and comparison of invariant records (containing no arrays as fields) for equality and inequality only.

```
<record type> ::= <invariant record type>
                        !<variant record type>
```

4.3.2.4.1 INVARIANT RECORDS

```
<invariant record type> ::=
    [packed] <invariant record type identifier>
   ![packed] <invariant record spec>

<invariant record type identifier> ::= <identifier>

<invariant record spec> ::=
          record <fixed fields> <recend>

<fixed fields> ::= <fixed field> {, <fixed field>}
<fixed field> ::= <field selectors> : [<alignment>] <fixed type>

<field selectors> ::= <field selector> {,<field selector>}
<field selector> ::= <identifier>

<recend> ::= [,]recend
```

See section 4.8 for a discussion on alignment.

4.3.2.4.2 VARIANT RECORDS AND CASE PARTS

A variant record consists of zero or more fixed fields followed by one and only one *case part*. A case part is a composite field that may assume values of different types during execution of a program. It is defined in terms of an optional *tag field*, and a list of the admissible types (called *variants*) together with associated *selection specs*. During execution, the value of the tag field may be used to determine the variant currently in use by being matched against the selection specs associated with each variant. The variants themselves may consist of zero or more fixed fields, optionally followed by one and only one case part.

Access to a variant other than the currently active variant produces undefined results. The currently active variation of a tagged variant record is the one associated with the current value of the tag field selector. The currently active variation of a tagless variant record is the one associated with the field that was the target of the last assignment to a field selector in the variations. Thus, the currently active variation changes when the tag field changes if there is a tag field or when an assignment is made to a field in a variation other than the currently active variation for tagless variant records. When this happens all fields in the newly active variation become undefined except for the target of the assignment for tagless variant records.

The space allocated for a variant record is the size of the largest variant regardless of which variant is used.

```
<variant record type> ::=
      [<packed>] <variant record type identifier>
     |[<packed>] <variant record spec>

<variant record type identifier> ::= <identifier>

<variant record spec> ::=
     record [<fixed fields>,] <case part> <recend>

<case part> ::= case <tag field spec> of
                          <variations><casend>

<tag field spec> ::= [<tag field selector> : ] <tag field type>
<tag field selector> ::= <identifier>
<tag field type> ::= <scalar type>

<variations> ::= <variation> {, <variation>}
<variation> ::= =<selection specs>= <variant>
```

4.0 CYBIL TYPES
4.3.2.4.2 VARIANT RECORDS AND CASE PARTS

```
<selection specs> ::= <selection spec>
                        {, <selection spec>}
<selection spec> ::= <constant scalar expression>
                   [..<constant scalar expression>]

<variant> ::= [<fixed fields>]
             ![<fixed fields>,] <case part>

<casend> ::= [,] casend
```

4.3.2.4.3 RECORD TYPE EQUIVALENCE

Two invariant record types are equivalent if they have the same packing, the same number of fields, and if corresponding fields have identical field selectors, the same alignment and equivalent types. Two variant record types are equivalent if they have identical tag field selectors and equivalent tag field types, the same number of variations and if variants having identical field selectors and equivalent types are selected by the same selection values.

4.3.2.4.4 PACKED RECORDS

Packing attributes are used to specify storage space versus access time tradeoffs for fields of records. Fields of packed records are mapped onto storage so as to conserve space at the possible expense of time. See section 4.7 and 4.8 for more details.

4.3.2.4.5 EXAMPLES OF RECORD TYPE

```
type
  date = record
           day : 1..31,
           month : string (4),
           year : 1900..2100,
         recend,

  status = record
             age : 6..66,
             married,
             sex : boolean,
           recend,

  red_book = record
               name : string (3),
               rstatus : status,
               scores : array[0..6] of date,
             recend,

  shape = (triangle, rectangle, circle),
  angle = -180..180,
```

4.0 CYBIL TYPES
4.3.2.4.5 EXAMPLES OF RECORD TYPE

```
figure = record x,
               y,
               area : real,{figure is a variant}
                           {record type}
         case s : shape of
         = triangle =
           size : real,
           inclination,
           angle1,
           angle2 : angle,
         = rectangle=
           side1,
           side2 : integer,
           skew,
           angle3 : angle,
         = circle =
           diameter: integer,
         casend,
         recend;
```

4.3.3 STORAGE TYPES

Storage types represent structures to which other variables may be added, deleted, and referenced under explicit program control.

```
<storage type> ::= <sequence type>
                  |<heap type>
```

4.3.3.1 Sequence_Type

```
<sequence type> ::= seq (<space>)
```

A sequence type represents a storage structure whose components are referenced (by a sequential accessing discipline) through pointers constructed as by-products of the next and reset operations. In addition, sequences may be assigned to sequences; no other operations are allowed.

Two sequences are equivalent if they have equivalent spaces.

4.3.3.2 Heap_Type

```
<heap type> ::= heap (<space>)
```

A heap type represents a structure whose components can be

explicitly allocated (by the allocate statement) and freed (by the free and reset statements), and which are referenced by pointers constructed as by-products of the allocate statement. No other operations on heaps are allowed.

Two heaps are equivalent if they have equivalent spaces.

A system-defined heap, that can be managed in the same manner as user-defined heaps, is provided.

4.3.3.3 Sequence_and_Heap_Space

```
<space> ::= <fixed span>{,<fixed span>}

<fixed span> ::=
     [rep <positive integer constant expression> of]
          <fixed type identifier>

<positive integer constant expression> ::=
     <constant scalar expression>

<fixed type identifier> ::= <identifier>
                          |<pre-defined type identifier>

<pre-defined type identifier> ::= integer | boolean | char | cell
```

A space attribute of the general form

```
rep n1 of type1, rep n2 of type2, ...
```

specifies a requirement that sufficient space be provided to simultaneously hold n1 instances of variables of type1, n2 instances of variables of type2, and so on.

Two spaces are equivalent if they have the same number of spans, and corresponding spans are equivalent. Two spans are equivalent if they have the same number of repetitions of equivalent types.

The space attribute places no restriction on the types of the variables that may be stored in a sequence or heap, other than that the space available for storage (as defined by the space attribute) be large enough to hold that many instances of the <fixed type identifier>. For example, the space attribute may be defined solely in terms of integers, but the sequence or heap filled only with strings of characters and boolean variables.

4.4 ADAPTABLE_TYPES

Adaptable types are structural skeletons of structured and storage types containing indefinite bounds, indicated by an asterisk. They may be used solely to define formal parameters of procedures and adaptable pointers, the latter providing a mechanism for referencing variables of such types.

Adaptable types represent classes of related types to which they can adapt. Adaptation to such an instantaneous type can occur in three distinct ways:

Adaptable types can be explicitly fixed by the use of allocation designators associated with storage management statements.

Adaptable types used as formal parameters are fixed by the actual parameters specified at procedure activation.

Adaptable pointer types used as left parts of assignment statements are fixed by the assignment operation.

```
<adaptable type> ::= <adaptable aggregate type>
                    !<adaptable storage type>

<adaptable aggregate type> ::= <adaptable string>
                              !<adaptable array>
                              !<adaptable record>

<adaptable storage type> ::= <adaptable sequence>
                            <adaptable heap>
```

4.4.1 ADAPTABLE STRING

Adaptable strings can adapt to strings of length 0 to 65535.

```
<adaptable string> ::= <adaptable fixed string>
                     ! <adaptable string identifier>

<adaptable fixed string> ::= string (<adaptable string length>)

<adaptable string length> ::= * ! * <= <adaptable string bound>

<adaptable string bound> ::= <length>

<adaptable string identifier> ::= <identifier>
```

If the adaptable string bound is not specified a string of maximum

allowable length is permitted.

In addition any string operation which exceeds the length specified by the adaptable string bound shall be an error and appropriate compile and run time checks will be included.

Two adaptable string types are always equivalent.

4.4.2 ADAPTABLE ARRAY

Adaptable arrays adapt to a specific range of subscripts.

Adaptable arrays can adapt to any array with the same packing and identical component type. If the lower bound is provided by the lower bound spec, the adaptable array can adapt only to arrays with an identical value for the lower bound.

```
<adaptable array> ::=
       [packed]<adaptable array identifier>
     ! [packed]<adaptable array spec>

<adaptable array identifier> ::= <identifier>

<adaptable array spec> ::=
      array [<adaptable array bound spec>] of <component type>

<adaptable array bound spec> ::= <lower bound spec> ..  *
                               ! *

<lower bound spec> ::= <constant integer expression>

<constant integer expression> ::= <constant expression>
```

The asterisk (*) indicates an adaptable bound of integer type.

Adaptable array types are equivalent if they have the same packing, and equivalent component types, and if corresponding array and component indices are equivalent. Two starred indices are always equivalent. Two starred indices with the lower bound spec selected are equivalent if their lower values are the same.

4.4.3 ADAPTABLE RECORD

Adaptable records consist of zero or more fixed fields followed by one and only one *adaptable field*, which is a field of adaptable type.

4.0 CYBIL TYPES
4.4.3 ADAPTABLE RECORD

Adaptable records can adapt to any record whose type is the same except for the type of its last field, which must be one to which the adaptable field can adapt.

```
<adaptable record> ::=
      [packed]<adaptable record type identifier>
     ![packed]<adaptable record spec>

<adaptable record type identifier> ::= <identifier>

<adaptable record spec> ::=
   record[<fixed fields>,]<adaptable field><recend>

<adaptable field> ::=
   <field selector>:[<alignment>]<adaptable type>
```

Two adaptable record types are equivalent if they have the same packing, the same alignment, the same number of fields, and corresponding fields have identical field selectors and equivalent types.

4.4.4 ADAPTABLE SEQUENCE

Adaptable sequences can adapt to a sequence of any size.

```
<adaptable sequence> ::= seq(*)
                        !<adaptable sequence identifier>

<adaptable sequence identifier> ::= <identifier>
```

The space for an adaptable sequence can be fixed by a <span fixer>.

Two adaptable sequence types are always equivalent.

4.4.5 ADAPTABLE HEAP

Adaptable heaps can adapt to a heap of any size.

<adaptable heap> ::= heap(*)
!<adaptable heap identifier>

<adaptable heap identifier> ::= <identifier>

The space for an adaptable heap can be fixed by a <span fixer>.

Two adaptable heap types are always equivalent.

4.5 PROCEDURE_TYPE

Procedures are identified portions of programs that can be activated on demand. Refer to chapters 8.0 and 10.0 for the semantics of procedures.

A procedure type defines an optional ordered list of formal parameters.

<procedure type> ::= <procedure type identifier>
!procedure <proc type spec>

<procedure type identifier> ::= <identifier>

Procedure types are used for declaration of pointers to procedures, there are no procedure variables.

Two procedure types are equivalent if corresponding param segments have the same number of formal parameters, identical methods (reference or value), and equivalent types.

4.6 BOUND_VARIANT_RECORD_TYPE

A bound variant record is a variant record whose case part is meant to be fixed to one of its constituent variants by the use of a tag field fixer. For bound variant records the <tag field selector> is required. These are space saving constructs, since only the space required for the selected variant is allocated.

Access to a variant other than the currently active variant produces undefined results. The currently active variation of a bound variant record is the one associated with the current value of

the tag field selector. Thus, the currently active variation changes when the tag field changes.

```
<bound variant record type> ::=
  [packed] <bound variant record type identifier>
 ![packed] bound <variant record spec>
 ![packed] bound <variant record type identifier>

<bound variant record type identifier> ::=
  <variant record type identifier>
```

A bound variant record type may only be used to define pointers for bound variant record types (i.e., bound variant pointers). Thus a variable of this type is always allocated in a sequence or a heap, or in the system-managed stack.

An allocation statement for a bound variant record type requires the specification of the tag field values, which select the variation of the record allocated. In this case, only the specified space is allocated. A bound variant pointer is returned by such an allocate statement.

If a formal parameter of a procedure is of variant record type, then the actual parameter may not be of bound variant record type.

Record assignment is not allowed to a variable of bound variant record type.

Two bound variant record types are equivalent if they are defined in terms of equivalent, unbound records. A bound variant record type is never equivalent to a variant record type.

4.7 PACKING

A packed structure will generally require less space at the possible cost of greater overhead associated with access to its components. If the packing attribute is unspecified, then the structure is assumed to be unpacked. An inner structure does not inherit the packing of any containing structure. Elements of packed structures are not guaranteed to lie on addressable memory units.

4.8 ALIGNMENT

```
<alignment> ::= aligned [[<offset> mod <base>]]

<offset> ::= <integer constant>
```

<base> ::= <integer constant>

The aligned attribute must be used to ensure addressability of fields within packed records. Addressability is achieved at the possible expense of storage space, so that the effect of packing may be diluted.

Unpacked structures and their components are always addressable. Packed structures are also addressable unless they are unaligned components of a packed structure, but their components are not unless they are explicitly given the aligned attribute. Aligning the first field of a record aligns the record.

A second usage of the alignment feature is to cause variables to be mapped onto a specified hardware address relative to a specified base and offset. The offset value must be less than the base and the base must be divisible by eight. The result is that an anonymous filler is created if necessary to ensure that the field begins on the specified addressable unit. For automatic variables, the base may only be eight. The <offset> and <base> elements are cell counts.

4.9 OTHER ASPECTS OF TYPES

4.9.1 VALUE AND NON-VALUE TYPES

Value assignments are permitted only to variables of the so-called value types. The non-value types are:

A) Heaps.
B) Arrays of non-value component types.
C) Records containing a field of non-value type.

4.9.2 COMPARABLE AND NON-COMPARABLE TYPES

Value comparisons are permitted only between variables of the so-called comparable types. The non-comparable types are:

A) Heaps.
B) Sequences.
C) Arrays.
D) Variant records.
E) Records containing a field of non-comparable type.

4.9.3 FUNCTION-RETURN TYPES

The only types that can be associated with returned values of functions are the basic types:

A) Integer, char, boolean, ordinal types, subrange types,
B) pointer types,
C) floating point types,
D) cell types.

4.9.4 TYPE CONVERSION

Mechanisms for converting values of some scalar types to values of others are provided.

A) Ordinal, character and boolean values are convertible to integer values through the ord function.

B) Integer values between 0 and 255 are convertible to characters by the chr function.

5.0 VALUES AND VALUE CONSTRUCTORS

5.0 VALUES_AND_VALUE_CONSTRUCTORS

Two mechanisms are provided for explicitly denoting values: constants and value constructors. Constants are used to denote constant values of the basic types and strings. Value constructors are used to denote instances of values of set, array and record types. There are two kinds of value constructors: set value constructors, which include specific type identification; and indefinite value constructors, whose type must be determined contextually.

5.1 CONSTANTS_AND_CONSTANT_DECLARATIONS

5.1.1 CONSTANTS

Constants are used to denote instances of values of the basic types and of string types.

```
<constant> ::= <basic constant>|<string constant>

<basic constant> ::= <scalar constant>
                   |<floating point constant>
                   |<pointer constant>

<scalar constant> ::= <ordinal constant>
                    |<boolean constant>
                    |<integer constant>
                    |<character constant>

<ordinal constant> ::= <ordinal constant identifier>

<boolean constant> ::= false | true
                     | <boolean constant identifier>

<boolean constant identifier> ::= <identifier>

<integer constant> ::= <integer> | <integer constant identifier>
```

5.0 VALUES AND VALUE CONSTRUCTORS
5.1.1 CONSTANTS

```
<character constant> ::= '<char token>'
                       ¦chr (<integer constant>)
                       ¦<character constant identifier>

<char token> ::= <alphabet>
               ¦ '' {two single quotes}

<character constant identifier> ::= <identifier>

<floating point constant> ::= <real constant>
               ¦ <longreal constant>

<real constant> ::= <real number> ¦ <real constant identifier>

<real constant identifier> ::= <identifier>

<real number ::= <unscaled number>
               ¦ <scaled number>

<unscaled number> ::= <digit> {<digit>}.  <digit>{<digit>}

<scaled number> ::= <mantissa> E<exponent>

<mantissa> ::= <digit>{<digit>}[.]{<digit>}

<exponent> ::= [<sign>]<digit>{<digit>}

<longreal constant> ::= <longreal number>
               ¦ <longreal constant identifier>

<longreal constant identifier> ::= <identifier>

<longreal number> ::= <mantissa> D<exponent>

<string constant> ::= <string term>
                          { cat <string term>}

<string term> ::= <character constant>
        ¦'[<char token> <char token> {<char token>}]'

<pointer constant> ::= nil

<integer constant identifier> ::= <identifier>

<integer> ::= <digit>{<digit>}
            ¦ <digit>{<hex digit>}<base designator>

<hex digit> ::= A¦B¦C¦D¦E¦F
              ¦a¦b¦c¦d¦e¦f
              ¦<digit>
```

5.0 VALUES AND VALUE CONSTRUCTORS
5.1.1 CONSTANTS

<base designator> ::= (<radix>)

<radix> ::= 2 | 10 | 16

If the base designator is omitted from an integer, then a radix of 10 is assumed. In all cases, the digits (or hex digits) are constrained to be less than the specified radix.

Note that string constants can be empty, that is, of zero length.

5.1.2 CONSTANT EXPRESSIONS

<constant scalar expression> ::= <constant expression>

<constant expression> ::= <simple expression>

Constant expressions are constructs denoting rules of computation for obtaining scalar or string type values (at compile time) by the application of operators to operands. The rules of application are those for expressions (see section 9) with the following constraints:

A) Factors of such expressions must be either constants or parenthesized constant expressions.

B) The expressions must be simple expressions (relational operators are not allowed).

C) The only functions allowed as factors in such expressions are the ord, chr, succ and pred.

D) Substring references are not allowed.

5.1.3 CONSTANT DECLARATIONS

Constant declarations are used to introduce identifiers for constant values. Once declared, such a constant identifier can be used elsewhere to stand for the identified value.

<constant declaration> ::=
    const <constant spec> {, <constant spec>}

<constant spec> ::= <identifier> = <constant expression>

A constant spec associates an identifier with the value and the type of the constant expression.

5.0 VALUES AND VALUE CONSTRUCTORS
5.2 SET VALUE CONSTRUCTORS

5.2 SET VALUE CONSTRUCTORS

Set value constructors are used to denote instances of values of a specified set type, and to denote instances of typed empty sets.

```
<set value constructor> ::=
     $<set type identifier> [ ]    {the empty set}
   ! $<set type identifier> [ <set value elements>]

<set value elements> ::= <set value element>
                         {,<set value element>}

<set value element> ::= <expression>
```

Identifiers for set value constructors are obtained by prefixing the 'target set type' identifier with a dollar sign, '$'. The types of the elements of the value constructor must match the ordered set of components of the specified target type. Set value constructors can be used wherever an expression can be used.

A set value element is an expression whose value is of the base type of the set. The elements of a set are unordered. Note that a set value may be defined to be 'empty' by not placing any elements between the brackets: [and].

5.3 INDEFINITE VALUE CONSTRUCTORS

Indefinite value constructors are used to denote instances of set, array, or record type.

```
<indefinite value constructor> ::=
                         [<value elements>]
                       ! [ ]   {the empty set}

<value elements> ::=
          <value element>{,<value element>}
<value element> ::= [<rep spec>]<initialization expression>
       ![<rep spec>]<set value constructor>
       ![<rep spec>]<indefinite value constructor>
       ![<rep spec>] *

<rep spec> ::= rep <positive integer expression> of
```

The meaning of a value constructor is that the list of values are assigned to the fields of a record or to the components of an array in their natural order. The types of the elements of the value constructor must match those of the components of the aggregate type

for which they provide the values.

Rep specs may be used solely for array construction, and indicate that the next n values are the same, as given by the value following the "OF".

Indefinite value constructors can be used only where their type is explicitly indicated by the context in which they occur: as elements of indefinite value constructors, and for the initialization of variables (see the discussion on Initialization in Section 6).

The asterisk form for a value element indicates that an undefined value may be assigned to the field or component at this position in the value list, unless it is a pointer in which case it is initialized to nil.

6.0 VARIABLES

6.0 VARIABLES

6.1 VARIABLES_AND_VARIABLE_DECLARATIONS

Variables take on values of a specific type (or range of types).

Variables of fixed type can be declared by an explicit variable declaration (see below) or can be declared as formal parameters of procedures.

Variables of adaptable type can only be declared as formal parameters of procedures, or must otherwise be explicitly established by storage management operations.

6.1.1 ESTABLISHING VARIABLES

This process involves:

A) The determination of the type of the variable;

B) The allocation of storage for values to be taken on by the variable;

C) The possible assignment of initial values to the variable;

D) The possible binding of references (see below) to that variable.

Locally declared variables are automatically established on each entry to the procedure block in which they were declared. However, so-called 'static' variables are established once and only once.

Formal parameters of procedures are automatically established on each call of that procedure.

So-called 'allocated' variables are established by storage management operations (for type determination and storage allocation) and by assignment operations (for initialization).

6.1.2 TYPING OF VARIABLES

Adaptable types and bound variant record types actually define classes of related types. Variables of such types (and pointers to such variables) are explicitly meant to be 'fixed' to any or all types of their type-class at different times during the execution of a program.

6.1.2.1 Instantaneous_Types

The type to which a variable is fixed at a specific time during execution of a program is called its instantaneous type (at that time). It is a variable's instantaneous type that is actually used to determine the operations it may enter into at any point in time.

Variables of adaptable and bound variant record type are fixed in three distinct ways:

A) Formal parameters of adaptable types are fixed by the instantaneous types of their corresponding actual parameters on each procedure call or function reference of which they are a part. (See Section 10.5.1 for the rules for fixing parameters.)

B) Explicitly allocated variables of such types are fixed by the allocation operation.

C) A pointer whose instantaneous type is any of the types to which an adaptable pointer can adapt, can be assigned to that adaptable pointer. In such cases, both the value and the type are assigned, thus fixing the instantaneous type of the adaptable pointer.

6.1.3 EXPLICIT VARIABLE DECLARATIONS

Variables are explicitly declared in terms of an identifier for denoting them, a type, an optional set of attributes and an optional initialization for static variables.

```
<variable declaration> ::=
    var <variable spec>
      {,<variable spec>}

<variable spec> ::=
      <variable identifiers> : [<attributes>]
      <fixed type>[<initialization>]

<variable identifiers> ::=
      <variable identifier> [<alias>]
      {,<variable identifier>[<alias>]}

<variable identifier> ::= <identifier>
```

6.2 ATTRIBUTES

```
<attributes> ::= [<attribute>{,<attribute>}]

<attribute> ::= <access attribute>
              |<storage attribute>
              |<scope attribute>
```

6.2.1 ACCESS ATTRIBUTE

```
<access attribute> ::= read
```

Variables declared with the read attribute are called 'read-only' variables. Such variables inherit the static attribute, must be initialized, may not be used as objects of assignment, and may be used as actual parameters only if the corresponding formal parameter is not a var parameter. The read attribute is used for compiler checking on access to variables and does not imply the variables residence in read-only storage on computer systems where that facility is provided. If the access attribute is not specified read and write access is implied.

Examples:

```
var     v1 : [read] integer := 10;  {v1 is read only, but
                                     {initialization is valid}
var     v2 : integer ; {v2 may be read and written}
```

6.2.2 STORAGE ATTRIBUTES AND LIFETIMES

<storage attribute> ::= static ! <section name>

Storage attribute specifies when storage for an explicitly declared variable is to be allocated (and initial values assigned if necessary) and when it is to be freed (at which time values of the variable become undefined). The programmatic domain in effect between the time such storage is allocated and the time it is freed is called the 'lifetime' of the variable.

6.2.2.1 Automatic_Variables

The lifetime of an automatic variable is the block in which it was declared: allocation occurs on each entry to that block and freeing occurs on each exit from that block. Variables not explicitly or implicitly declared static have the automatic attribute.

6.2.2.2 Static_Variables

The lifetime of a static variable is the entire program: allocation and initialization occur once and only once (at a time not later than initial entry to the block in which the variable was declared), and storage is not freed on exits from that block.

6.2.2.3 Lifetime_Conventions

If neither storage attributes nor scope attributes are specified, then the variable is treated as an automatic variable.

If the static attribute is specified then the variable is treated as a static variable.

If any of the scope attributes are specified, then the variable is treated as a static variable.

Variables declared at the outermost level of a module body are treated as static variables.

6.2.2.4 Lifetime_of_Formal_Parameters

The lifetime of a formal parameter is the lifetime of the procedure of which it is a part: the formal parameter is established on each entry to the procedure, and becomes undefined on exits from

the procedure.

6.2.2.5 Lifetime_of_Allocated_Variables

Allocated variables are established (but not initialized, except in the case of tag fields of bound variant records) by an explicit allocation operation, and become undefined when they are explicitly freed.

6.2.2.6 Pointer_Lifetimes

Warning: Note that generally a pointer value has a finite lifetime different from that of the pointer variable. Automatic variables cease to exist on exit from the block in which they were declared. Allocated variables cease to exist when they are freed or when their containing variable ceases to exist. Attempts to reference non-existent variables by a designator beyond their lifetime is a programming error and could lead to disastrous results. Failure to free a variable allocated via an automatic pointer before the containing procedure returns will prevent space for that variable from ever being released by the program.

6.2.3 SCOPE ATTRIBUTES

<scope attribute> ::= xdcl ! xref ! #gate

Variable identifiers are used in variable denotations. Scope attributes specify the regimen to be used to associate instances of variable identifiers with instances of variable specs. The programmatic domain over which a variable spec is associated with instances of its associated variable identifiers that are used in variable denotations, is called the scope of that spec. If no scope attribute is specified, the spec is said to be internal to the procedure or function block in which it occurs, and a so-called block-structuring regimen is used.

Internal variables are always automatic variables (see above) unless given a storage attribute, while scope-attributed variables are always static. Each of the scope attributes specifies certain deviations from the block-structuring regimen. Broadly speaking, a variable identifier associated with an xref variable can be used to denote a similarly identified variable having the xdcl attribute, subject only to reasonable rules of specificational conformity.

Xref variables can not be initialized, and each carries the de-facto static storage attribute.

For more details on scope attributes, see section 7.

There should exist only one declaration of a given variable or procedure with the xdcl attribute within a compilation unit or within a group of compilation units to be combined for execution.

The #gate attribute is an extension of the xdcl attribute to extend the protection provided for in the environment provided by the operating system. It may not be relevant on all computer systems. Specifying the #gate attribute without also specifying xdcl is a compilation error.

6.3 INITIALIZATION

Initializations are used to specify values to be assigned to static variables.

```
<initialization> ::= := <initialization expression>

<initialization expression> ::= <constant expression>
                              ! <indefinite value constructor>
                              ! ^<global proc name>

<global proc name> ::= <procedure identifier>
```

When the variable is established, the type of the variable is determined, storage for a variable of that type is allocated as a static variable, the initialization expression is evaluated, and the resultant value is assigned to the variable according to the normal rules for assignment.

6.3.1 INITIALIZATION CONSTRAINTS

1) If no initialization is specified, the initial value is undefined, except that all pointer components of static variables are initialized to nil.

2) If the initialization expression is an indefinite value constructor, the variable must be either a set, array, or record. The type of the indefinite value constructor is determined as the type of the variable.

3) An asterisk, '*', can be used in indefinite value constructors to indicate uninitialized elements of arrays and records. The initial values of such uninitialized elements are undefined, except in the case of a pointer which is set to nil.

4) If the string elements are not of equal length and the variable part is the longer, the initialization operator will append blanks at the right end of the variable. If the initialization expression is longer, the value of the initialization expression will be truncated to fit the variable part.

5) Within variant record initialization, the case selector is initialized in turn and is then used to determine the variant for the ensuing fields of the record.

6.4 SECTIONS_AND_SECTION_DECLARATIONS

A section is a working storage area for specified variables sharing common access attributes.

<section declaration> ::= section <sections> {,<sections>}

<sections> ::=
<section name> {,<section name>} : <section attribute>

<section name> ::= <identifier>

<section attribute> ::= read | write

Variables declared within a section having the read section attribute will reside in read-only storage (on computer systems providing that facility) and must have the read variable attribute.

6.5 VALID_COMBINATIONS_OF_ATTRIBUTES_AND_INITIALIZATIONS

Only certain combinations of attributes are valid. These combine with certain initialization assignments, some of which are optional, some required, and some prohibited.

The table below further clarifies the legal combination of attributes and specifies the rules for initialization.

6.0 VARIABLES
6.5 VALID COMBINATIONS OF ATTRIBUTES AND INITIALIZATIONS

	ATTRIBUTE	INITIALIZATION	SAME_AS
(1)	none	optional if static otherwise prohibited	
(2)	read	required	(4)
(3)	static	optional	
(4)	static,read	required	(2)
(5)	xdcl	optional	(7)
(6)	xdcl,read	required	(8)
(7)	xdcl,static	optional	(5)
(8)	xdcl,static,read	required	(6)
(9)	xref	prohibited	(11)
(10)	xref,read	prohibited	(12)
(11)	xref,static	prohibited	(9)
(12)	xref,static,read	prohibited	(10)
(13)	<section name>	optional	*
(14)	<section name>,read	required	*
(15)	<section name>,xdcl	optional	*
(16)	<section name>,xdcl,read	required	*

* Static attribute is implied for sections.

6.6 VARIABLE_REFERENCES

```
<variable> ::= <variable reference>
             !<substring reference>

<variable reference> ::= <variable identifier>
                       !<pointer reference>^
                       !<subscripted reference>
                       !<field reference>
```

6.6.1 POINTER REFERENCES

<pointer reference> ::= <pointer variable>
                      ;<function reference>

<pointer variable> ::= <variable>

Whenever a variable reference denotes a variable of pointer type, it is referred to as a pointer reference and the notation

<pointer reference>^

may be used to denote a variable whose type is determined by the type associated with the pointer variable. If another variable of pointer type is denoted by this reference, then

<pointer reference>^^

may be used as a variable reference. Note that variables of pointer type can be components of structured variables as well as valid return types for functions.

Given a variable identifier, the notation to obtain a pointer value to the variable which has a scope equal to or greater than the pointer is:

^<variable identifier>

Pointers are always bound to a specific type and pointer variables may assume, as values, only pointers to objects of equivalent type. The only exception to this is that pointer to cell can take on values of any type and any fixed or bound variant pointer variable can assume a value of pointer to cell. See Chapter 4 for further explanation.

The special value nil is used to denote that a pointer variable has no current assignment to a location.

6.6.1.1 Examples of Pointer References

```
var i, j, k : integer, {integer variables}

    pi : ^integer, {pointer variable of type: pointer to integer}

    ppi : ^^integer, {pointer variable of type:}
                     {pointer to pointer to integer}

    b1, b2 : boolean ; {boolean variables--end of declarations}
```

6.0 VARIABLES
6.6.1.1 Examples of Pointer References

```
allocate pi; {allocates space for an integer value and sets}
         {pi to point to it}

allocate ppi; {allocates space for a pointer to integer and}
          {sets ppi to point to it}

pi^ := 10;

ppi^ := pi;

j := pi^ ; {the integer variable j takes on the value 10}

k := ppi^^ ; {the integer variable k takes on the value 10

b1 := j = k ; {the boolean variable takes on the value true}

b2 := pi^ = ppi^^ ; {the boolean variable b2 takes on the
                              {value true}

pi := nil ; {the pointer variable pi is set to denote
                 {lack of indicating any variable}

k := pi^ ; {statement is in error when pi has the
             {value nil--result of this statement
             {will be implementation dependent}

if ppi = nil then k := k + 1 ifend ;
  {valid test of ppi and valid statement}

pi := ^(i + j + 2 * k); {improper use of up arrow to request}
            {location of an expression - an undefined concept}
```

6.6.2 SUBSTRING REFERENCES

```
<substring reference> ::=
        <string variable>(<substring spec>)
<string variable> ::= <variable reference>

<substring spec> ::=
        <first char>[,<substring length>]

<first char> ::= <positive integer expression>
<substring length> ::= <non-negative integer expression>
                     ; *

<non-negative integer expression> :: = <scalar expression>
```

Values of string variables are ordered n-tuples of character values. Substring references yield fixed or null strings defined as

follows.

Let 's' denote a string whose current length is n.

If 1 <= i <= n then:

A) 's(i)' yields a fixed string of length one, consisting of the i-th character of s;

If 1 <= i <= n + 1 and 0 <= k <= n + 1 - i, then:

B) 's(i,k)' yields a fixed string of length k, consisting of the i-th through the (i+k-1)-th character of s, or a null substring;

C) 's(i,*)' is equivalent to 's(i,n-i+1)' and yields the rest of the string starting with the i-th character, or a null string.

Otherwise, an error results.

6.0 VARIABLES
6.6.2 SUBSTRING REFERENCES

Example:

If a string variable s is declared and initialized by

```
var s : string(6) := 'ABCDEF';
```

then the following relations hold

```
s(1) = 'A'      s(2,5) = 'BCDEF'
s(6) = 'F'      s(2,*) = s(2,5)
s(1,6) = s      s(1,*) = s
s(2,0) = ''     s(7,*) = ''
```

and s(8) and s(8,0) are illegal.

If a pointer variable is declared and initialized by:

```
var ps : ^string (6) := ^s;
```

then ps^(i) and ps^(i,j) become valid references to substrings of s.

Note that a string constant, even if declared with an identifier for denoting it, is not a variable, so that a substring of such a string constant is not a defined entity of CYBIL, e.g.,

```
const str24 = 'helper';

     ...

string2 := str24(3,*) ; {invalid substring reference--str24}
                        {is a string constant}
```

6.6.2.1 Substring_References_as_Character_References

Substring references of the form 's(k)', and only such, may be used wherever a character expression is allowed, and are treated as characters in such cases. Specifically, substrings of the form 's(k)':

A) May be compared with characters;

B) May be tested for membership (in) in sets of characters;

C) May be used as initial and final values of for statements controlled by a character variable;

D) May be used as selectors in case statements;

E) May be used as arguments of the standard procedures and functions

succ, pred, and ord;

F) May be assigned to character variables, and may be actual parameters to formal parameters of character type.

G) May be used as index values corresponding to character-type indices.

6.6.3 SUBSCRIPTED REFERENCE

<subscripted reference> ::= <array variable> [<subscript>]

<array variable> ::= <variable>

<subscript> ::= <scalar expression>

A subscripted reference denotes a component of an array variable, whose value type is the component type of the array variable. A subscript may be of any type that can be assigned to a variable of the corresponding index type. Note that, to this end, any subrange is considered to be of equivalent type as its parent range (or any subrange thereof).

Example:

If an array variable is declared and initialized by:

```
var A : array [1..5] of integer := [1, 2, 3, 4, 5]
```

and an integer variable is declared and initialized by

```
var i : integer := 5
```

then the following relations hold

```
a[i]   = 5
a[i-1] = 4
     .
     .
     .
a[i-4] = 1
```

However, the reference a[i+1] would be in error.

If an array variable is declared by:

```
var b: array [0..5] of array [0..9] of char
```

then b[1][2] becomes a valid reference to the array b.

If a pointer variable is declared and initialized by:

```
var pa : ^array [1..5] of integer := ^a;
```

then pa^[i] becomes a valid reference to components of a.

6.6.4 FIELD REFERENCES

```
<field reference> ::=
   <variable reference>[.<record subreference>]

<record subreference> ::=
   <field selector>|<subscripted reference>
```

A field reference denotes a field of a record variable. Since field selector names can be used in other records, the record variable must be specified.

Example:

For the record variable declared and initialized by

```
   type tr = record age : 6..66,
                    married,
                    sex : boolean,
                    date : record day : 1..31,
                                  month : 1..12,
                                  year : 70..80,
                           recend,
             recend;

    var r : tr := [23,false,true,[3,5,73]];
```

the following relations hold

```
   r.age = 23
   r.married = false
   r.sex = true
   r.date.day = 3
   r.date.month = 5
   r.date.year = 73
```

If a pointer variable is declared and initialized by:

```
     var pr : ^tr := ^r
```

then

```
     pr^.age, pr^.married, ...
```

become valid references to fields of tr.

7.0 PROGRAM STRUCTURE

7.0 PROGRAM_STRUCTURE

7.1 COMPILATION_UNITS

A CYBIL program is a collection of declarations which is meant to be translated, via a compilation process, into a CYBIL object module. Object modules resulting from separate compilations can be combined, via a linking process, into a single object module, and may undergo further transformations into a form capable of direct execution.

```
<compilation unit> ::= <module declaration>
                       {;<module declaration>} [;]
```

Since statements are constrained to appear solely within the body of a procedure or function declaration, compilation units consist solely of a list of declarations. All such declarations must be capable of being evaluated at the time of compilation. All variables declared in a compilation unit's declaration list will automatically be given the static storage attribute.

7.2 MODULES

A module is a collection of declarations.

```
<module declaration> ::=
     module <module identifier> [<alias>];
          <module body>
     moduend [<module identifier>]

<module identifier> ::= <identifier>
<module body> ::= <declaration list>

<declaration list> ::= {<declaration>;}
```

The module identifier can be used to provide clarity and to assist in post-compilation activities, such as linking and debugging.

7.3 DECLARATIONS AND SCOPE OF IDENTIFIERS

Declarations introduce objects together with identifiers which may be used to denote these objects elsewhere in a program.

```
<declaration> ::= <type declaration>
                | <constant declaration>
                | <variable declaration>
                | <procedure declaration>
                | <function declaration>
                | <section declaration>
                | <empty>
```

The programmatic domain over which all uses of an identifier are associated with the same object is called the scope of the identifier. Within a compilation unit, such a programmatic domain is either a module body, a procedure body or a function body. The scope of an identifier is determined by the context in which it was declared and by optional scope attributes which may be associated with declarations of variables and procedures.

7.4 MODULE - STRUCTURED SCOPE RULES

The scope of an identifier declared in one of the constituent declarations of the body of a module, is the body of that module.

7.5 PROCEDURES AND FUNCTIONS

A procedure or a function consists of a statement list preceded by an optional declaration list. Procedures and functions have three purposes:

1) Procedures and functions control the scope of identifiers.

2) Unlike modules, procedures and functions control the processing of declarations and determine when declarations take effect.

3) Unlike modules, procedures and functions include statements, which translate into algorithmic actions in the resulting program.

7.6 STRUCTURED_SCOPE_RULES

1) Except for field selectors (see below), the scope of an identifier declared in the constituent declaration list of a procedure or function is the body of that procedure or function.

2) If an identifier labels a structured statement then its scope is that statement.

3) If the scope of an identifier includes a non-xrefed procedure or function declaration, then its scope is extended 'downward' to include the body of that procedure or function, unless the body includes a re-declaration of the identifier.

4) The scope of an identifier which is declared as a formal parameter of a procedure or function is the body of the procedure or function.

5) Field selectors are identifiers introduced as part of the declaration of a record type for purposes of selecting fields of records. Except for the restriction that field selectors associated with the same record type must be unique, identifiers used as field selectors may be re-declared with impunity.

6) Except for field selectors, no more than one declaration of an identifier can be included in the constituent declarations and statements of the body of a procedure or function.

7.7 SCOPE_ATTRIBUTES

The scope attributes xdcl and xref cause the scope of identifiers to be extended, in a discontinous manner, to include other compilation units, but do not otherwise contravene either module-structured or block-structured scope rules.

Variables, procedures and functions that are part of one module, but are meant to be referenced from other modules, must have the xdcl attribute associated with them by explicit declaration. Other modules which are meant to reference such objects must declare them with the xref attribute.

XREF variables can not be initialized, and all xdcl and xref variables are automatically given the static storage attribute

The declarations for objects shared among modules must match; for example, an identifier with the xdcl attribute in one module and the xref attribute in other modules must denote the same object in all

such modules. Violations of such matching rules are detected during the linking processing on some computer systems.

7.7.1 ALIAS NAMES

An 'alias' is an alternate spelling which may be specified for an identifier. Its reasons for existence are varied: to meet system-requirements of spelling which are invalid in CYBIL, to equate two differing spellings for an entity between two different compilation units, to avoid identifier spelling conflicts among different compilation units or with system standard names, etc. As such, this feature will only be supported on host systems where this requirement exists.

An alias is to be used outside of a compilation unit only, and will not function as an alternative spelling for an identifier within the compilation unit in which it is defined as an alias.

Aliases may be furnished for identifiers of modules, procedures, and variables by following the identifier associated with a declaration of such an object by an alias specification.

<alias> ::= alias ' <alphabet> { <alphabet> } '

In order for an alias to 'reach' the host system, it must be associated with an object that is externalized in some way: by virtue of being xref'd, or xdcl'd. All other aliases will be inoperative except for taking up room during the compilation process.

If an identifier which is externalized has an alias specified, then only the alias will be made known outside of the compilation unit (i.e., the identifier itself will not be made known outside of the compilation unit).

Also refer to 6.1 for variable declarations, and to 8.1 for procedure declarations.

Examples:

```
module outer alias 'CYM$OUT' ; ...

procedure [xdcl] searcher alias 'CYP$SEARCH' (var lst2,...

var V2 alias 'CYV$2FLAG', V3 alias 'CYV$3FLAG' : [xdcl ] integer;
```

7.8 DECLARATION PROCESSING

7.8.1 BLOCK-EMBEDDED DECLARATIONS

Except for the constituent declarations of a compilation unit (see below), declaration processing is governed solely by block-structure. During compilation, all constituent lists of a block are gathered together and are processed en-masse, all such declarations coming into effect simultaneously.

Block-structure also governs declaration processing during execution of the resulting programs. On entry to a block, all declarations included in the block's constituent list are again collected together, storage for automatic variables is allocated, and all identifiers declared by such declarations become accessible. On exit from a block, all identifiers declared within that block become inaccessible, the values of automatic variables become undefined, and the variables allocated on the system-managed stack become undefined.

7.8.2 MODULE-LEVEL DECLARATIONS

Objects declared at the outermost level of a module are associated with no block at all. Such declarations must be evaluated, and required storage allocated, prior to program execution. Accordingly, all variables so declared are automatically given the static storage attribute, as are all scope-attributed variables.

8.0 PROCEDURES AND FUNCTIONS

8.0 PROCEDURES AND FUNCTIONS

A procedure or function declaration defines a portion of a program and associates an identifier with it so that it can be activated (i.e., executed) on demand by other statements in the language. A procedure or function is invoked by a procedure call statement or function reference.

A procedure call statement or function reference causes the execution of the constituent declarations and statement lists of the procedure or function after substituting the actual parameters of the call for the formal parameters of the declaration.

8.1 PROCEDURE DECLARATIONS

There are the following forms of procedure declaration:

```
<procedure declaration> ::=
        procedure [ xref ] <proc spec>
      ! procedure[[<proc attribute>]]<proc spec>;
                <proc body><proc end>
      ! program <proc spec>;<proc body><proc end>
```

The first form is used to refer to a procedure which has been compiled as part of a different module. The procedure must have been declared with the xdcl attribute, and with an equivalent parameter list in that module.

The second and third forms declare the procedure identifier to be a procedure of the kind specified by its parameter list and associates the identifier with the constituent declaration list and statement list of the declaration.

The program declaration is used to identify the first procedure of a program to be executed, when required by the system. It may only be present on a single outermost block level procedure of the compilation unit.

If more than one compilation unit is to be linked together for execution, then only one procedure with a program declaration may be present among all those compilation units being linked.

The procedure type is elaborated on entry to the block in which it

8.0 PROCEDURES AND FUNCTIONS
8.1 PROCEDURE DECLARATIONS

is declared, and remains fixed throughout the execution of that block.

```
<proc attribute> ::= xdcl ! #gate, xdcl ! xdcl, #gate

<proc spec> ::= <procedure identifier> [<alias>] <proc type spec>

<proc type spec> ::= [<parameter list>]
<parameter list> ::= (<param segment> {;<param segment>})
<param segment> ::= <reference params>
                  ! <value params>
<reference params> ::= var <param> { ,<param> }
<param> ::= <formal param list> : <parameter type>
<value params> ::= <value param>{,<value param>}
<value param> ::= <formal param list> :
                  <parameter type> [:= <default>]
<default> ::= <constant>

<formal param list> ::= <formal parameter identifier>
                        {,<formal parameter identifier>}

<formal parameter identifier> ::= <identifier>

<parameter type> ::= <fixed type>
                     !<adaptable type>

<proc body> ::= <declaration list> <statement list>

<proc end> ::= procend [<procedure identifier>]

<procedure identifier> ::= <identifier>
```

The default value provides an optional parameter capability for the value parameters on xref'ed procedures. Reference parameters may not be optional. See section 10.5 for a discussion on procedure calls involving optional parameters.

The #gate attribute is an extension of the xdcl attribute to extend the protection provided for in the environment provided by the operating system. It may not be relevant on all computer systems. Specifying the #gate attribute without also specifying xdcl is a compilation error.

8.2 FUNCTION DECLARATIONS

```
<function declaration> ::= function [ xref ] <func spec>
          ! function [[ func attribute]] <func spec> ;
                     <func body> <func end>
```

8.0 PROCEDURES AND FUNCTIONS
8.2 FUNCTION DECLARATIONS

<func spec> ::= <function identifier> [<alias>]<func type spec>

<function identifier> ::= <identifier>

<func type spec> ::= [<parameter list>] : <result type>

<result type> ::= <basic type>

<func attribute> ::= <proc attribute>

<func body> ::= <proc body>

<func end> ::= funcend [<function identifier>]

Function declarations serve to define parts of the program which compute a value of the basic type. Functions are activated by the evaluation of a function reference which is a constituent of an expression.

The value of a function is the value last assigned to its function identifier before returning (either by falling through the funcend, by a return statement, or by an exit statement). The results of returning by any means from a function prior to assignment of a value to the function identifier (for the current execution) are undefined.

8.2.1 SIDE EFFECTS

A function returns a value through the identifier of the function. When a function changes the value of a variable other than the local variables of the function that change is a side effect. CYBIL prevents side effects by restricting assignments, procedure and function calls, and the use of non-local variables in user defined functions.

The left-hand side of an assignment statement within a function may not be any of the following:

- A non-local variable,
- A reference parameter of the function,
- A pointer variable followed by a dereference (^).

User defined functions may not contain:

- Procedure call statements that call user-defined procedures,
- Parameters of type pointer to procedure,
- ALLOCATE, FREE, PUSH or NEXT statements that have parameters

that are not local variables.

These restrictions may make it necessary to use a procedure for some purposes for which a function might otherwise be used. However this inconvenience may provide more reliability by preventing side effects.

8.3 XDCL_PROCEDURES_AND_FUNCTIONS

The attribute **xdcl** may only be used on a procedure or function declared at the outermost level; i.e., not contained in another procedure or function. It specifies that the procedure or function should be made referenceable from other modules which have a declaration for the same procedure or function identifier with the **xref** attribute. The parameters must also be the same.

8.4 PARAMETER_LIST

A parameter list is a set of variable declarations in the <proc type spec> or <func type spec> (not in the <proc body>) which provides a mechanism for the binding of references to the procedure or function call environment in a manner which permits selection of entities to be bound at each invocation of the procedure or function. This is accomplished by providing the procedure or function with a set of values and variables, so-called actual parameters, at the point of call.

A value parameter results in the value of the actual parameter, at the point of call, being associated with the formal parameter. See section 10 for precise rules governing parameter passing. The called procedure or function may not assign a value to one of its value parameters, nor use it as an actual reference parameter to any procedure or function it may call.

The type of a formal value parameter may be any fixed or adaptable type except the so-called non-value types: heaps, records and arrays of non-value types (i.e., any type which cannot enter into an assignment statement may be neither a formal nor an actual value parameter).

A reference parameter results in the formal parameter designating the corresponding actual parameter throughout execution of the procedure. Assignments to the formal parameter thus cause changes to the variable that was passed as the corresponding actual parameter.

The type of a formal reference parameter may be any fixed or

adaptable type.

Examples:

```
procedure gcd (m, n : integer ; var x, y, z : integer) ;

var a1, a2, b1, b2, c, d, q, r : integer ; {m > 0,n > 0}
      {Greatest Common Divisor x of m and n,
      {Extended Euclid's Algorithm.}

      a1 := 0 ;
      a2 := 1 ;
      b1 := 1 ;
      b2 := 0 ;
      c := m ;
      d := n ;

      while d <> 0 do
        {a1 * m + b1 * n = d, a2 * m + b2 * n = c
        {gcd(c, d) = gcd(m, n)}

        q := c div d ;
        r := c mod d ;
        a2 := a2 - q * a1 ;
        b2 := b2 - q * b1 ;
        c := d ;
        d := r ;
        r := a1 ;
        a1 := a2 ;
        a2 := r ;
        r := b1 ;
        b1 := b2 ;
        b2 := r ;
      whilend ;

      x := c;
      y := a2;
      z := b2;
      {x = gcd(m, n), y * m + z * n = gcd(m, n)}
procend gcd;
```

```
function min (a: integer; b: integer): integer;

  if a > b then
    min := b;
  else
    min := a;
  ifend;

funcend min;
```

9.0 EXPRESSIONS

9.0 EXPRESSIONS

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands (i.e., variables and constants), operators, and functions.

Constant expressions are expressions involving constants and a subset of the operators and functions (cf., Section 5).

```
<expression> ::= <simple expression>
         ¦<simple expression><relational operator>
                       <simple expression>

<simple expression> ::= <term> ¦ <sign operator><term>
                        ¦<simple expression>
                           <adding operator><term>

<term> ::= <factor>
        ¦ <term><multiplying operator><factor>

<factor> ::= <variable>¦<constant>¦<constant identifier>
     ¦<set value constructor>¦<function reference>
     ¦^<procedure identifier>¦^<variable>
     ¦(<expression>)¦not<factor>

<multiplying operator> ::= * ¦ div ¦ / ¦ mod ¦ and
<sign operator> ::= <sign>
<sign> ::= + ¦ -
<adding operator> ::= + ¦ - ¦ or ¦ xor
<relational operator> ::= < ¦ <= ¦ > ¦ >= ¦ = ¦ <> ¦ in
```

9.0 EXPRESSIONS

```
<function reference> ::= <built-in function reference>
                        !<user defined function reference>

<user defined function reference> ::=
          <function identifier>(<actual parameter>
          {, <actual parameter>})
          ! <function identifier>()

<built-in function reference> ::= succ (<scalar expression>)
          !pred (<scalar expression>)
          !ord (<expression>)
          !chr (<expression>)
          !$integer (<expression>)
          !$real (<expression>)
          !$longreal (<expression>)
          !strlength (<fixed string type identifier>
                     !<string variable>)
          !lowerbound (<fixed array type identifier>
                      !<array variable>)
          !upperbound (<fixed array type identifier>
                      !<array variable>)
          !uppervalue (<scalar type identifier>
                      !<scalar variable>)
          !lowervalue (<scalar type identifier>
                      !<scalar variable>)
          !#loc (<variable>)
          !#size(<variable>
                !<fixed type identifier>
                !<adaptable type> : [<adaptable field fixer>])

<fixed string type identifier> ::= <string type identifier>

<fixed array type identifier> ::= <array type identifier>

<fixed type identifier> ::= <identifier>

<scalar type identifier> ::= <scalar identifier>

<scalar variable> ::= <variable>
```

See Section 11 for the details of these built-in functions.

9.0 EXPRESSIONS

Examples:

Factors:

```
x
15
(x + y + z)
$colorset [red, c, green]
not p
```

Terms:

```
x * y
i div 3
p and q
(x <= y) and (y < z)
```

Simple expressions:

```
x + y
- x
bool1 or bool2
i * j + 1
hue - $colorset [red, green]
```

Expressions:

```
x = 1
p <= 2
(i<j) = (j<k)
c in huel
```

9.1 EVALUATION_OF_FACTORS

The value of a variable, as a factor, is the value last assigned to it as possibly modified by subsequent assignments to its components.

The value of an unsigned number is the value of type integer denoted by it in the specified radix system.

The value of a real or longreal constant is the number denoted by it.

String constants consisting of a single character denote the value of type char of the character between the quote marks.

String constants of n (n > 1) characters denote the fixed string (n) value consisting of the characters between the quote marks.

The constant nil denotes a null pointer value of any pointer type.

9.0 EXPRESSIONS
9.1 EVALUATION OF FACTORS

A constant identifier is replaced by the constant it denotes. If this in turn is a constant identifier, the process is repeated until a constant of one of the above forms results. The value is then obtained as above.

The value of a set value constructor is the value obtained from the values of its constituent expressions of type specified by its set type identifier.

The value of an up-arrow followed by a variable of type T is the pointer value that designates that variable.

The value of an up-arrow followed by a procedure identifier of procedure type P is the pointer to procedure value that designates the current instance of declaration of that procedure.

A function reference specifies the execution of a function. The actual parameters are substituted for the corresponding formal parameters in the declaration of the function. The body is then executed. The value of the function reference is the value last assigned to the function identifier. The meaning of, and restrictions on, the actual parameters is the same as for the procedure call statement (see 10.5.1).

The value of a parenthesized expression is the value of the expression which is enclosed by the parentheses.

The type of the value of a factor obtained from a variable or function reference whose type is a subrange of some scalar type is that scalar type.

9.2 OPERATORS

Operators perform operations on a value or a pair of values to produce a new value. Most of the operators are defined only on basic types, though some are defined on most types. The following sections define the range of applicability, as well as result, of the defined operators. An operation on a variable or component which has an undefined value will be undefined in result.

9.2.1 NOT OPERATOR

The not operator, not, applies to factors of type boolean. When applied the meaning is negation; i.e., not true = false and not false = true.

9.0 EXPRESSIONS
9.2.2 MULTIPLYING OPERATORS

9.2.2 MULTIPLYING OPERATORS

The following table shows the multiplying operators, the types of their permissible operands, and the type of the result.

Operator	Operation	Operands	Result
*	multiplication	integer or integer subrange	integer
		real	real
		longreal	longreal
	set intersection - the set consisting of elements common to the two sets	set of type T	set of type T
div	integer quotient for a, b, n positive integers a div b = n where n is the largest integer such that b*n < = a for one or two negative integers (-a) div b = (a) div (-b) = - (a div b), a div b = (-a) div (-b)	integer or integer subrange	integer
/	real and longreal quotient	real longreal	real longreal
mod	remainder function a mod b = a - (a div b)*b	integer or integer subrange	integer
and	logical 'and' true and false = false true and true = true false and false = false false and true = false *When the first operand is false, the second is never evaluated.	boolean	boolean

9.2.3 SIGN OPERATORS

The + operator can be applied to integer, real and longreal types only. For types integer, real and longreal it denotes the identity operation and results in integer, real or longreal type (i.e., a = + a).

The - operator can be applied to integer, real, longreal and set types only. It denotes sign inversion--i.e., -a = 0 - a for integers, reals or longreals. It denotes complementation for sets with respect to the base type - i.e., the set of all elements of the base type not contained in the specified set.

9.2.4 ADDING OPERATORS

The following table shows the adding operators, the types of their permissible operands, and the type of the result.

9.0 EXPRESSIONS
9.2.4 ADDING OPERATORS

Operator	Operations	Operands	Result
+	addition	integer or integer subrange	integer
		real	real
		longreal	longreal
	set union - the set consisting of all elements of both sets.	set of type T	set of type T
-	subtraction	integer or integer subrange	integer
		real	real
		longreal	longreal
	boolean difference true - true = false, true - false = true false - true = false, false - false = false	boolean	boolean
	set difference - the set consisting of elements of the left operand that are not also elements of the right operand.	set of type T	set of type T
or	logical 'or' true or true = true, true or false = true false or true = true, false or false = false * When the first operand is true, the second is never evaluated.	boolean	boolean
xor	exclusive 'or' true xor true = false true xor false = true false xor true = true false xor false = false	boolean	boolean
	symmetric difference - the set of elements	set of type T	set of type T

9.0 EXPRESSIONS
9.2.4 ADDING OPERATORS

	contained in either set but not both sets.			

9.2.5 RELATIONAL OPERATORS

Relational operators are the primary means of testing values in CYBIL. They yield the boolean value true if the specified relation holds between the operands, and the value false, otherwise.

9.2.5.1 Comparison_of_Scalars

All six comparison operations < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), = (equal to), and <> (not equal to) are defined between operands of the same scalar type, or substrings of length one and char.

For operands of type integer they have their usual meaning.

For operands of type boolean the relation false < true defines the ordering.

For operands, a and b, of type char, the relation a op b holds if and only if the relation ord(a) op ord(b) holds, where op denotes any of the six comparison operators and ord is the mapping function from character type to integer type defined by the ASCII collating sequence.

For operands of any ordinal type T, a = b if, and only if, a and b are the same value; a < b if, and only if, a precedes b in the ordered list of values defining T.

Operands of type subrange of some parent scalar type may be compared with operands whose type is the parent type or another subrange of that parent type.

9.2.5.2 Comparison_of_Pointers

Two pointers can be compared if they are pointers to either equivalent or potentially equivalent types. In the latter case, one or both of the pointers may be pointers to adaptable or bound variant types. The instantaneous type of such pointers must be equivalent to the type of the pointer they are being compared with; if it is not, the operation is undefined.

Pointers may be compared for equality and inequality only.

A pointer of any type may be compared for equality or inequality with the value nil.

A pointer comparison results in equality if both pointers

designate the same variable, or if they both have the value nil.

Two pointers to procedure are equal if they designate the same instance of declaration of a procedure.

9.2.5.3 Comparison of Floating Point Types.

All six relations are defined between operands of real and longreal types, respectively. Comparison for equality and inequality is done within the precision limits of the host machine.

9.2.5.4 Comparison of Strings

All six relational operators may be applied to operands whose values are strings. If the actual lengths of the two strings entering into the operation are unequal, blanks are conceptually appended to the string having the shorter length.

Strings are compared to each other character by character from left to right until total equality or inequality is determined, as follows. Let n be the length of the strings a and b ($n \geq 1$), and op be any of the six comparison operators, then:

- a = b iff a(i) = b(i) for all $1 \leq i \leq n$
- For op one of <>, <, >

  a op b iff for some k, $1 \leq k \leq n$
  a(k) op b(k) AND
  a(i) = b(i) for $1 \leq i < k$

- a >= b iff a = b OR a > b
- a <= b iff a = b OR a < b

9.2.5.5 Relations Involving Sets

The relation a in s is true if the scalar value a is a member of the set value s. The base type of the set must be the same as, or a subrange of, the type of the scalar, or the scalar type may be a subrange of the base type of the set.

The set operations = (identical to), <> (different from) <= (is included in), and >= (includes) are defined between two set values of the same base type.

s1 = s2 is true if all members of s1 are contained

in s2, and all members of s2 are contained in s1.

s1 <> s2 is true when s1 = s2 is false.

s1 <= s2 is true if all members of s1 are also members of s2.

s1 >= s2 is true if all members of s2 are also members of s1.

9.2.5.6 Relations Involving Arrays and Records

1) Arrays may never be compared. Structures which contain an array as component or field may never be compared.

2) Variant records can not be compared. Other record types may be compared for equality or inequality only. Two equivalent records are equal if and only if corresponding fields are equal.

9.2.5.7 Non-Comparable Types

Certain types in the language cannot be compared. These are heaps, sequences, arrays, variant records, and records containing a field of a non-comparable type. However, pointers to non-comparable types can be compared.

9.2.5.8 Table of Comparable Types and Result Types

The following table shows the relational operators, the types of their permissible operands, and the type of the result.

Operator	Operation	Left Operand	Right Operand	Result
< <= > >= = <>	- less than - less than or equal to - greater than - greater than or equal to - equal to - not equal to	any scalar type T	T' where T and T' are comparable	boolean
		string(n) S(k) * char	string(n) char S(k) *	boolean boolean boolean
in	set membership test	any scalar type T	set of T' where T' and T are comparable	boolean
		S(k) *	set of char type	boolean
= <> <= >=	- identity - different - is contained in - contains	set of T where T is any scalar type	set of T	boolean
= <>	- equal to - not equal to	any non-variant record type T containing no arrays	T (the same type)	boolean
		any pointer type T or nil	T or nil	boolean

(*) Substring of form S(k) with a length of one implied.
The form S(k,1) is not legal in these contexts.

9.3 ORDER_OF_EVALUATION

The rules of composition specify operator precedence according to five classes of operators. The not operator has the highest precedence, followed by the multiplying operators, followed by the sign operators, then the adding operators, and finally, with the lowest precedence, the relational operators.

The precise order in which the operands entering into an expression are evaluated is only partially defined. The order of application of operators is defined by the composition rules (and their implied hierarchy of operator precedence) with the exception that the order of application is undefined for any sequence of commutative operators of the same precedence class. For example:

1) The expression a * b * c div d is evaluated as (a * b * c) div d, and the internal order of evaluation of the first term is undefined.

2) The expression a + b + c - d is evaluated as (a + b + c) -d, with the internal order of evaluation of (a + b + c) undefined.

3) In the evaluation of boolean expressions, terms and factors are evaluated from left to right, and evaluation terminates as soon as the value of the term or expression is determined.

10.0 STATEMENTS

10.0 STATEMENTS

Statements denote algorithmic actions, and are said to be executable. A statement list denotes an ordered sequence of such actions. A statement is separated from its successor statement by a semicolon. The successor to the last statement of a statement list is determined by the structured statement or procedure of which it forms a part.

```
<statement list> ::= <statement>{;<statement>}

<statement> ::= <assignment statement>
              ;<structured statement>
              ;<control statement>
              ;<storage management statement>
```

10.1 SEMICOLONS AS STATEMENT LIST DELIMITERS

Since the successor of the last statement of a statement list is uniquely determined by the structured statement or procedure of which it is a part, semicolons are not required as statement list delimiters. However, since the empty statement is allowed, semicolons may be so used for consistency of presentation.

10.2 ASSIGNMENT STATEMENTS

The assignment statement is used to replace the current value of a variable by a new value derived from an expression.

<assignment statement> ::=
<variable> := <expression>

10.2.1 ASSIGNMENT COMPATIBILITY OF TYPES

The part to the left of the assignment operator (:=) is evaluated to obtain a reference to some variable. The expression on the right is evaluated to obtain a value. The value of the referenced variable is replaced by the value of the expression.

The variable on the left may be of any data type except for:

- Any variable specified as read-only, or a formal value parameter of any containing procedure.
- Any bound variant record.
- The tag field of any bound variant record.
- Heaps, and arrays and records containing heaps.

The variable or function identifier on the left and the expression on the right must be of equivalent instantaneous type, except as noted below:

- The types of the variable and the expression may be subranges of an equivalent parent type, or one may be a subrange of the other. If the value of the expression is not a value of the type of the variable, the program is in error.
- If the left part is a character variable, a string variable or a substring, the expression may be a character expression, a string or a substring.
- If the strings, substrings or character elements are not of equal length and the destination part (left part) is the longer, the assignment operator will append blanks at the right end of the destination variable. If the source part (right part) is longer, the assignment will truncate the value of the source part on the right to fit the destination part.
- Assignment of two substrings which overlap one another is not

allowed and the results are unpredictable.

- If the left part is a variant record, the right part may be a bound variant record of otherwise equivalent types.

- If the left part is a pointer, its scope must be less than or equal to the scope of the data to which it is pointing. For example, a static pointer variable cannot point to a local variable.

- If the left part is a pointer to a bound variant record, the expression may be a pointer to an otherwise equivalent 'unbound' variant record.

- If the left part is an adaptable pointer, the right part must be either a pointer to any of the instantaneous types to which the left part pointer can adapt, or an adaptable pointer which has been adapted to one of those types. Both the type of the expression and its value are assigned, thus setting the current type of the assignee.

- If the left part is a fixed or bound variant pointer type, the right part may be a pointer to cell. The only effect of the assignment is as follows: after the assignment, the value returned by an application of the #loc function on the de-referenced value of the lefthand side as argument will be equal to the right-hand side value.

- If the left part is a pointer to cell, the right part may be a pointer type. The value assigned is a pointer to the first cell allocated for the variable pointed-to by the right side.

- Warning: Note that generally a pointer value has a finite lifetime (see Section 6.2.2) different from that of the pointer variable. Automatic variables cease to exist on exit from the block in which they were declared. Allocated variables cease to exist when they are freed. Attempts to reference non-existent variables by a designator beyond their lifetime is a programming error and could lead to disastrous results.

10.3 STATEMENT LABELS

A structured statement may be labeled by preceding it with a structured statement identifier. This allows the statement to be explicitly referred to by other constituent statements (e.g., exit, cycle). Such a labeling of a statement constitutes the declaration of the structured statement identifier and hence the identifier must differ from all other identifiers declared in the same block.

10.3.1 SCOPE OF STRUCTURED STATEMENT IDENTIFIERS

If a structured statement identifier labels a constituent structured statement of a procedure declaration or a begin statement, then its scope is that procedure declaration or begin statement. It is impossible to refer to a structured statement identifier on a structured statement from outside that statement. A structured statement identifier may optionally follow a structured statement (except repeat.. until), in which case it must be identical to the structured statement identifer labeling that statement. This is for checking purposes only, and does not affect the meaning of the program. The scope of a structured statement identifier does not include procedures called from within its scope.

```
<structured statement identifier> ::= /<identifier>/
```

Example:

```
/check_range/
  while val < 0 do
        .
        .
        .
  whilend /check_range/;
```

10.4 STRUCTURED STATEMENTS

Structured statements are constructs composed of statement lists. They provide scope control, selective execution, or repetitive execution of their constituent statement lists.

```
<structured statement> ::= [<structured statement identifier>]
                             <repeat statement>
          ![<structured statement identifier>] <delimited statement>
                             [<structured statement identifier>]

<delimited statement> ::= <begin statement>
          ! <while statement>
          ! <for statement>
```

10.4.1 BEGIN STATEMENTS

Begin statements permit the execution of a single statement list. Exit is either through completing execution of the last statement of the statement list or through an explicit transfer of control.

10.0 STATEMENTS
10.4.1 BEGIN STATEMENTS

The successor of the last statement of the statement list of a begin statement is the successor of the begin statement.

```
<begin statement> ::=
     begin <statement list> end
```

10.4.2 WHILE STATEMENTS

A while statement controls repetitive execution of its constituent statement list.

```
<while statement> ::=
     while <expression> do <statement list> whilend
```

The expression controlling repetition must be of type boolean. The statement list is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement list is not executed at all.

The successor of the last statement of the constituent statement list of a while statement is the while statement itself.

Examples:

```
while a[i] <> x do
  i := i + 1;
whilend;

while i > 0 do
  if i = z then
    z := z * x;
  ifend;
  i := i div 2;
  x := x * x;
whilend;
```

10.4.3 REPEAT STATEMENTS

A repeat statement controls repetitive execution of its constituent statement list.

```
<repeat statement> ::=
     repeat <statement list> until <expression>
```

10.0 STATEMENTS
10.4.3 REPEAT STATEMENTS

The expression controlling repetition must be of type boolean. The statement list between the symbols repeat and until is repeatedly (and at least once) executed until the expression becomes true.

Example:

```
repeat
  k := i mod j;
  i := j;
  j := k;
until  j = 0;
```

10.4.4 FOR STATEMENTS

The for statement indicates that its constituent statement list is to be repeatedly executed while a progression of values is assigned to a variable, which is called the control variable of the for statement.

```
<for statement> ::=
          for <control variable> := <for list> do
               <statement list> forend
<for list> ::=
      <initial value> to <final value>
     |<initial value> downto <final value>

<control variable> ::= <variable identifier>
<initial value> ::= <scalar expression>
<final value> ::= <scalar expression>
<scalar expression> ::= <expression>
```

The control variable, initial value and final value must all be of equivalent scalar type or subranges of equivalent types.

The control variable may not be an unaligned component of a packed structure.

Assignment to the control variable, either explicit or by passing as a var parameter, within the statement list is a fatal compilation error.

The initial value and final value are evaluated once on entry to the for statement, as is the name of the control variable. Thus, subsequent assignments to components of these expressions have no effect on the sequencing of the statement.

10.0 STATEMENTS
10.4.4 FOR STATEMENTS

If the initial value is greater than the final value in the to form, or if the initial value is less than the final value in the downto form, then no assignment is made to the control variable and the statement list is not executed.

If the exit from the statement is a normal one, then the value of the control variable is the final value. If the exit is caused by the exit statement, the value of the control variable is that which was in effect when the exit statement was executed.

10.5 CONTROL_STATEMENTS

Control statements cause the transfer of control to a different execution environment or to a statement other than the successor statement in the same environment, or both.

```
<control statement> ::= <procedure call statement>
          ! <if statement> ! <case statement>
          ! <cycle statement>
          ! <exit statement> ! <return statement>
          ! <empty statement>
```

10.5.1 PROCEDURE CALL STATEMENT

A procedure call statement causes the creation of an environment for the execution of the specified procedure and transfers control to that procedure. (cf., Chapter 8.0 Procedures.) A procedure call statement may never be used to activate a function.

```
<procedure call statement> ::=
     <procedure reference> <actual parameter list>

<procedure reference> ::= <procedure identifier>
                        ! <pointer to procedure reference> ^

<pointer to procedure reference> ::= <pointer reference>

<actual parameter list> ::=
       (<actual parameter>{,<actual parameter>})
     ! (<keyword parameter> {,<keyword parameter>})
     ! <empty>

<keyword parameter> ::=
     <formal parameter name> := <actual parameter>

<empty> ::=
```

10.0 STATEMENTS
10.5.1 PROCEDURE CALL STATEMENT

<actual parameter> ::= <expression>
: <empty>

An actual parameter corresponds to the formal parameter which occupies the same relative position in the formal parameter list. When using the keyword form, actual parameters may be specified in any order. However, no procedure call can use part keyword and part positional form. In either form, parameters which are optional (formal read parameters with a default) may be omitted by using an empty positional parameter or by not being specified in the keyword form. In either case, the value given the omitted parameter is that declared as the default value. With these default parameters, commas used to mark out to the end of the parameter list are unnecessary.

10.5.1.1 Value_Parameters

A value parameter causes the association within the called procedure of the value of the actual parameter at the point of call with the name of the formal parameter. The type of the parameter is fixed as follows:

1) If the formal parameter is of fixed type, then the actual parameter may be any expression which could be assigned to a variable of that type, except in the case of strings which must be of equal length.

2) If the formal parameter is of adaptable type, the instantaneous type of the actual parameter must be one of those to which the adaptable type can adapt.

3) If the formal parameter is an adaptable pointer, then the actual parameter may be any pointer expression which could be assigned to that adaptable pointer. Both the value and the instantaneous type of the actual parameter are assigned, thus fixing the type of the formal parameter.

10.5.1.2 Reference_Parameters

A var parameter causes the formal parameter to designate the actual parameter throughout execution of the procedure. Assignments to the formal parameter thus cause changes to the corresponding actual parameter. An actual parameter corresponding to a var formal parameter must be addressable.

The type designated by the formal parameter is fixed as follows:

1) If the formal parameter is of fixed type, the actual parameter

must be a variable or substring reference of equivalent type.

2) If the formal parameter is of adaptable type, the actual parameter must be a variable or substring reference whose type is potentially equivalent.

10.5.2 IF STATEMENTS

The if statement provides for the execution of one (and only one) of a set of statement lists depending on the value of boolean expression(s). The boolean expression(s) following the if or elseif symbols are evaluated in order until one is found whose value is true. The subsequent statement list is then executed.

If the value of all Boolean expression(s) are false, then either no statements are executed, or the statement list following the else symbol is executed (if present).

The successor to the last statement of a constituent statement list of an if statement is the successor of the if statement.

```
<if statement> ::=
    if<if body> ] ifend

<if body> ::= <expression> then <statement list>
    [ else <statement list> ! elseif <if body>]
```

Examples:

```
if x < y then
  x := y;
ifend;

if x <= 5 then
  z := 1;
elseif x > 30 then
  z := 2;
elseif x = 15 then
  z := 3;
else
  z := 4;
ifend;
```

In the first example, x takes on the value of y if and only if the relation x < y holds>. In the second example, z will take on one of the values (1,2,3,4) depending on the value of x.

10.5.3 CASE STATEMENTS

A case statement selects one of its component statement lists for execution depending on the value of the selector expression.

```
<case statement> ::= case <selector> of <cases>
                     [else <statement list>] casend

<selector> ::= <scalar expression>

<cases> ::= <a case>{;<a case>}
<a case> ::= =<selection spec>{,<selection spec>}=
                <statement list>

<selection spec> ::=
             <constant scalar expression>
          [..<constant scalar expression>]
```

The case statement selects for execution that statement list (if any) which has a selection specification which includes the value of the selector. If no selection specification includes the value of the selector, the statement list following else is selected when the else option is employed. If the value of the selector is not included in any selection spec and the else is omitted, the program is in error.

The selector and all selection specifications must be of the same scalar type or subranges of the same type. No two selection specifications may include the same values (i.e., selection must be unique).

Selection specs are restricted to simple constant scalar expressions.

The successor of the last statement of a selected statement list is the successor of the case statement.

10.0 STATEMENTS
10.5.3 CASE STATEMENTS

Examples:

```
case operator of
  =plus=   x := x + y ;
  =minus=  x := x - y ;
  =times=  x := x * y ;
casend;

case i of
  =1=      x := x+1 ;
  =2=      x := x+2 ;
  =3=      x := x+3 ;
  =4=      x := x+4 ;
else
  x := -x ;
casend;

type lextype = (basic, inconst, realconst, stringconst,
   identifier),
  symbol = record
    case lex : lextype of
    =basic=
      name : symbolid,
      class : operation,
    =inconst=
      value : integer,
      optimiz : boolean,
    =realconst=
      value : real,
    =stringconst=
      length : 1..255,
      stringbuf : ^string(*),
    =identifier=
      identno : integer,
      decl : ^symbolentry,
    casend;
  recend;
var
  cursym : symbol,
  sign : boolean := false;
insymbol;
case cursym.lex of
  =basic=
    if cursym.name= minus then
      sign := not sign;
    else
      error ('missing operand');
    ifend;
  =inconst=
    cursym.optimiz := (cursym.value<halfword);
```

```
      if sign then
        sign := false;
        cursym.value := -cursym.value;
      ifend;
    =realconst=
      if sign then
        sign := false;
        cursym.value := -cursym.value;
      ifend;
    =stringconst=
      error ('string constant where arithmetic type expected');
    =identifier=
      cursym.decl := symbolsearch;
      if cursym.decl^.typ <> constdecl then
        variable (cursym.decl);
      else
        cursym := cursym.decl^.value^;
      ifend;
  casend;
```

10.5.4 CYCLE STATEMENT

The cycle statement allows the conditional by-passing of the remainder of the statements of the constituent statement list of the designated repetitive statement, causing reevaluation of the expression controlling the structured statement, thus cycling it to its next iteration (if any).

<cycle statement> ::= cycle <structured statement identifier>

The structured statement identifier must identify a repetitive statement (for, while, or repeat statement), which statically encompasses the cycle statement, i.e., the cycle statement must be within the scope of the structured statement.

Thus, the cycle statement has the effect of (potentially) re-executing the statement list of a repetitive statement such as for, repeat, or while.

Examples:

```
    x := a[1];
  /find_smallest/
    for k := 2 to n do
      if x < a[k] then
        cycle /find_smallest/;
      ifend;
      x := a[k]; {this assignment skipped when x < a[k]}
```

10.0 STATEMENTS
10.5.4 CYCLE STATEMENT

```
        {this finds the smallest value in a[1] thru a[n]}
    forend /find_smallest/;
```

10.5.5 EXIT STATEMENT

The exit statement causes execution to continue at the successor of a designated structured statement, procedure or function..

```
<exit statement> ::= exit <structured statement identifier>
                       ! <procedure identifier>
                       ! <function identifier>
```

If a procedure or function identifier is designated as the object of the exit, then that procedure or function must statically encompass the exit statement within the same module. If a structured statement identifier is designated as the object of the exit, then that identifier must be for a structured statement which statically encompasses the exit statement within the same module.

Note that the exit statement with either a structured statement identifier, procedure identifier or function identifier designated permits multiple levels of exit with a single statement. Thus, exit can permit recursive nests to be terminated with a single statement by selection of the appropriate structured statement, procedure or function identifier.

10.0 STATEMENTS
10.5.5 EXIT STATEMENT

Examples:

```
/meaningful_label/
  begin  x := y + 27 ; {example of exit <label>}
    found := false; ...
  /for_while_loop/
    for k := 1 to 10000 do
      j := k ;
      if (i mod 2) = 0 then
        b[k] := false;
      else
        prime(i, answer) ; {test if prime}
        while true do
          if answer = 5 then
            exit /for_while_loop/; {goes to 'bound := j;' statement}
          ifend ;
          answer := answer - 5 ;
          if answer <= 0 then
            exit /meaningful_label/; {exit: while, for
            {and begin stmt and goes to ' if found then ...'}
          ifend;
        whilend;
      ifend;
    forend /for_while_loop/;
    {exit /for_while_loop/ causes control to transfer here}
    bound := j;
    found := true;
  end /meaningful_label/;
  {exit /meaningful_label/; causes control to transfer here}
  if found then ...  ;
```

10.5.6 RETURN STATEMENT

The return statement causes the current procedure or function to return i.e. completes the current activation of the procedure or function.

<return statement> ::= return

10.5.7 EMPTY STATEMENT

An empty statement denotes no action and consists of no symbols.

<empty statement> ::=

10.6 STORAGE_MANAGEMENT_STATEMENTS

There are two storage types, sequences and heaps, defined in the language, each with its own unique management and access characteristics. Variables of such types define structures into which other variables may be placed, referenced, and deleted under program control according to the discipline implied by the type of the storage variable. Storage management statements are the means for effecting this control, and for managing the placement of variables into the so-called system-stack.

```
<storage management statement> ::= <push statement>
                                 ¦<next statement>
                                 ¦<reset statement>
                                 ¦<allocate statement>
                                 ¦<free statement>
```

10.6.1 ALLOCATION DESIGNATOR

An allocation designator specifies the type of the variable to be managed by the storage management statements. An allocation designator is either:

A) A pointer to a fixed type, in which case a variable of the type designated by the pointer variable is specified;

or

B) An adaptable pointer (or bound variant record pointer) followed by a type fixer (see below) which specifies the adaptable bounds, lengths, sizes, or tag fields, in which case a variable of the resultant fixed type is designated and the adaptable or bound variant record pointer is set to designate a variable of that type.

```
<allocation designator> ::=
  <fixed pointer variable>
 ¦<adaptable array pointer variable> : [<star fixer>]
 ¦<adaptable string pointer variable> : [<length fixer>]
 ¦<adaptable storage pointer variable> : [<span fixer>]
 ¦<adaptable record pointer variable> : [<adaptable field fixer>]
 ¦<bound variant record pointer variable> :
     [<tag field fixers>]

<fixed pointer variable> ::= <pointer variable>

<adaptable array pointer variable> ::= <pointer variable>
```

10.0 STATEMENTS
10.6.1 ALLOCATION DESIGNATOR

```
<adaptable string pointer variable> ::= <pointer variable>

<adaptable storage pointer variable> ::= <pointer variable>

<adaptable record pointer variable> ::= <pointer variable>

<bound variant record pointer variable> ::= <pointer variable>

<tag field fixers> ::= <scalar expression>
          ! <constant fixers>[,<scalar expression>]

<constant fixers> ::= <constant scalar expression>
                    {,<constant scalar expression>}

<adaptable field fixer> ::= <star fixer>
                          !<length fixer>
                          !<span fixer>

<star fixer> ::= <scalar expression> .. <scalar expression>
<length fixer> ::= <positive integer expression>
<span fixer> ::= [<span> {, <span> }]
<span> ::= [rep <positive integer expression> of]
     <fixed type identifier>
```

1) Star fixers are used in the fixing of adaptable bounds of arrays. Values for both the lower and upper bound must be specified in the star fixer. If the lower bound was provided by a lower bound spec, the corresponding value specified in the star fixer must be identical to the value specified by the lower bound spec.

2) Length fixers are used in the fixing of adaptable bounds of strings.

3) Span fixers are used in the fixing of adaptable bounds of heaps or sequences.

4) The type and value of an adaptable field fixer must select one of the types to which the associated adaptable pointer can adapt.

5) The order, types, and values of tag field fixers must select those variants to which the associated bound variant record pointer can be bound. All but the last of these tag field fixers must be constant expressions.

6) For the bounds list used in an allocation designator, entries are required only for the dimenion which is adaptable.

7) Pointers associated with type fixers are set to designate a variable of the type fixed by the type fixer (whenever the statement in which they occur is executed). They will then designate a variable of that fixed type until they are either reset by a subsequent assignment operation or re-fixed by a type fixer in a subsequent storage management operation.

Example:

```
type
  tipe = array[1..*] of array [1..5] of array [10..20]
          of array [21..24] of integer ;
var
  point : ^tipe ,
  bunch : heap (rep 25000 of integer) ;
        {point is an adaptable pointer variable}
          ...
reset bunch;
          ...
allocate point : [1..15] in bunch ;
```

This allocate statement would cause the allocation of an array of four dimensions with components of type integer, with dimensions:

1 to 15, 1 to 5, 10 to 20, and 21 to 24.

and would set the pointer variable, point, to designate that array.

10.6.2 PUSH STATEMENT

The push statement causes the allocation of space for a variable on the system-managed stack (system stack), and sets an allocation designator to designate that variable (or to the pointer value nil if there is insufficient space for the allocation). The value of the newly allocated variable (or of any component thereof, in the case of structured variables) remains undefined until the subsequent assignment of a value to the variable or to its components.

<push statement> ::= push <allocation designator>

10.6.2.1 System-Managed_Stack

A variable allocated on the system-stack can not be explicitly de-allocated by the user. Instead, de-allocation occurs automatically on exit from the procedure containing the allocating push statement, at which time space for the variable is released and its value becomes undefined.

Example:

```
var localarray : ^array[1..*] of integer ;
push localarray :[1..20];

{allocate space for array [1..20] of integer on
{system stack, i-th element can be referenced
{as localarray^[i]}
```

10.6.3 NEXT STATEMENT

The next statement sets the allocation designator to designate the current element of the sequence, and causes the next element to become the current element. This results in the positioning information in the variable of type pointer to sequence to be updated. After a reset or an allocation of a sequence, the current element is the first element of the sequence. Note that the ordered set of variables comprising a sequence is determined algorithmically by the sequence of execution of next statements.

The type of the pointer variable must be the same when the data is retrieved from a sequence as when that same data was stored into the sequence; otherwise, the program is in error.

If the execution of a next statement would cause the new current element to lie outside the bounds of the sequence, then the

allocation designator is set to the value nil.

<next statement> ::=
    next <allocation designator> in <pointer to sequence variable>

<pointer to sequence variable> ::= <pointer variable>

Example:

```
next length_ptr in buf_ptr ;
next stgptr : [1..length_ptr^] in buf_ptr ;
```

10.6.4 RESET STATEMENT

The reset statement causes either positioning in a sequence, or en-masse freeing of all variables of a heap. Space for freed variables is released and their values become undefined.

<reset statement> ::=
    reset <pointer to sequence variable> [to <pointer variable>]
  | reset <heap variable>

Warning: a reset statement is required prior to the first allocate statement for any user-defined sequence or heap to reset the sequence or heap to an 'empty' status; otherwise, the program is in error.

10.6.4.1 Reset_Sequence

The reset sequence statement causes the positioning information contained in a variable of type pointer to sequence to be reset. If the optional to clause is not specified, the first element of the sequence becomes the current element of the sequence. If the to is specified, the element in the sequence pointed to by the <pointer variable> becomes the current element of the sequence. The use of a pointer variable whose value had not been set by a next statement for the same sequence, or whose value is nil, is an error.

10.6.4.2 Reset_Heap

The reset heap statement causes all elements currently allocated in the specified heap to be freed en-masse.

10.6.5 ALLOCATE STATEMENT

The allocate statement causes the allocation of a variable of the specified type in the specified heap and sets the allocation designator to designate that variable or to the value nil if there is insufficient space for the allocation. If a heap variable is not specified, the allocation takes place out of the universal (system defined) heap.

Note that the first allocate statement for any heap (other than the system heap) must be preceded by the execution of a reset statement for that heap, or the program will be in error.

```
<allocate statement> ::= allocate <allocation designator>
          [ in <heap variable>]

<heap variable> ::= <variable>
```

Examples:

```
var my_array : ^array [*] of integer;

allocate my_array : [0..49]; {allocate space in system heap}
allocate sym_ptr in symbol_table;
```

10.6.6 FREE STATEMENT

The free statement causes the deletion of a specified variable from the specified heap or from the system heap if the in clause is omitted: space for the variable is released, and its value becomes undefined.

A pointer variable specifies the variable to be freed. If the variable specified is not currently allocated in the heap, the effect is undefined. Execution of the free statement sets the pointer variable to the value nil. Use of a pointer variable with a value of nil to attempt data access is an error. Freeing a nil pointer is an error.

10.0 STATEMENTS
10.6.6 FREE STATEMENT

<free statement> ::=
    free <pointer variable>[in <heap variable>]

Examples:

```
free sym_ptr in symbol_table;
free my_array;
```

11.0 STANDARD PROCEDURES AND FUNCTIONS

11.0 STANDARD_PROCEDURES_AND_FUNCTIONS

Certain standard procedures and functions have been defined for CYBIL which have been included because of the assumed frequency of their use or because they would be difficult or impossible to define in the language in a machine-independent way.

11.1 BUILT-IN_PROCEDURE

11.1.1 STRINGREP (S, L, P)

In this procedure, S is a <string variable>, L is a <result length>, and P is a scalar expression.

The string representation procedure facilitates the conversion of P to its representation as a string of characters.

The value of P is converted into a string of characters. The resulting string is returned, left-justified, in the <string variable> S. The <result length> L returned is an integer variable whose value is the length (in characters) of the result string.

If the expression to be converted is an integer expression, the resultant string shall be in the base 10. If the integer expression is negative in value, then a minus sign precedes the leftmost significant digit within the field. If positive, then a blank character precedes the integer value. If the field given is not long enough to contain all the digits of the value of the integer expression, then the output field is filled with a string of asterisk characters.

If the expression to be converted is an ordinal expression, then the integer value of the ordinal is handled in exactly the same manner as an integer element.

If the expression to be converted is a boolean expression, then the five character string ' TRUE' or 'FALSE' is placed left justified into the output field with a length of 5. If the field length given is not long enough to contain all five characters, then the output field is filled with a string of asterisk characters.

11.0 STANDARD PROCEDURES AND FUNCTIONS
11.1.1 STRINGREP (S, L, P)

The conversion rules for floating point are to be defined.

11.2 BUILT-IN FUNCTIONS

The following standard functions return values of the specified type.

11.2.1 SUCC(X)

The type of the expression, x, must be scalar, and the result is the successor value of x if it exists; if not, the program is in error.

11.2.2 PRED(X)

The type of the expression, x, must be scalar, and the result is the predecessor value of x if it exists; if not, the program is in error.

11.2.3 ORD(X)

Returns the integer representation of the value x. The type of the expression, x, must be ordinal, char, or boolean.

If x is boolean then zero (0) is returned for false and one (1) for true. If x is char, the value returned is the ordinal number, in the ASCII collating sequence, of x. If x is an ordinal constant, the value returned is the ordinal number of that constant.

11.2.4 CHR(X)

X must be an integer expression yielding a value $0 \leq x \leq 255$. The value returned is the character whose ordinal number in the ASCII collating sequence is x.

11.2.5 $INTEGER(X)

Returns the integer value corresponding to the value of x. The type of the expression, x, must be ordinal, char, boolean, integer or subrange of integer, real or longreal. The conversions are done as follows:

a) if X is ordinal, the value returned is the ordinal number of the ordinal constant identifier associated with the ordinal value;

b) if X is character, the value returned is the ordinal number, in the ASCII collating sequence, of the value of X;

c) if X is boolean, zero (0) is returned for false and one (1) for true;

d) if X is an integer value that value is returned;

e) if X is a real or longreal value, that value is first truncated to a whole number. If the resultant value is within the range of type integer, then that value is returned, otherwise, an out-of-range error occurs.

11.2.6 $REAL(X)

Returns the real number which is the implementation dependent approximation of the integer or longreal expression. In the case of a longreal, the most significant part is returned. Longreals are truncated as part of the conversion.

11.2.7 $LONGREAL(X)

Returns a longreal result which is the implementation dependent approximation of the integer or real expression.

11.2.8 STRLENGTH(X)

Returns the length of the string x. For a fixed string this is the allocated length, and x may be either a string variable or a string type identifier. For an adaptable string this is the current length and x must be an adaptable string reference.

11.2.9 LOWERBOUND(ARRAY)

Returns the value of the low bound of the array index. The type of the result is the index type of the array. The argument (array) may be either an array variable or a fixed array type identifier.

11.2.10 UPPERBOUND(ARRAY)

Returns the value of the upper bound of the array index. The type of the result is the index type of the array. The argument (array) may be either an array variable or a fixed array type identifier.

11.2.11 UPPERVALUE (X)

Accepts as argument either a scalar type identifier or a variable of scalar type. It returns the largest possible value which an argument of that type can take on. The type of the result is the type of x.

11.2.12 LOWERVALUE (X)

Accepts as argument either a scalar type identifier or a variable of scalar type. It returns the smallest possible value which an argument of that type can take on. The type of the result is the type of x.

11.3 REPRESENTATION_DEPENDENT_FUNCTIONS

11.3.1 #LOC(<VARIABLE>)

Returns a pointer to the first cell allocated for the specified variable.

11.3.2 #SIZE(ARGUMENT)

Returns the number of cells required to contain the variable, or a variable of the argument type. The argument may be either a variable

or a fixed, adaptable or bound variant type identifier. In the case of adaptable type identifier the adaptable field fixer must also be specified. In the case of the bound variant type identifier, the variant requiring the largest size is the value returned.

11.4 SYSTEM_DEPENDENT_PROCEDURES

The capability to generate certain C180 instructions inline is provided by the following general form:

#INLINE ('name', p1, ...pn)

where name is the identifier for the set of instructions to be generated.

11.4.1 #INLINE ('KEYPOINT', P1, P2, P3)

Causes the keypoint instruction to be generated inline based on the following parameters:

p1 - is a constant expression in the range 0..15 and becomes the instructions J field,

p2 - is a constant or variable expression, if it is the constant zero then the K field of the instruction is zero, if not zero or a variable then that value is placed in an X register and that register becomes the instruction's K field,

p3 - is a constant expression in the range 0..0FFFF(16) and becomes the instruction's Q field.

12.0 COMPILE-TIME FACILITIES

12.0 COMPILE-TIME FACILITIES

Compile-time facilities are essentially extra-linguistic in nature in that they are used to construct the program to be compiled and to control the compilation process, rather than having a meaning in the program itself. These, together with commentary and programmatic elements of the language, are the elements of a CYBIL source text.

12.1 CYBIL SOURCE TEXT

```
<text> ::= <text item> {<text item>}

<text item> ::= <pragmat statement>
              ! <compile-time statement>
              ! <identifier>
              ! <constant>
              ! <basic symbol other than ??>
              ! <comment>

<compile-time statement> ::= <compile-time declaration>
                           ! <compile-time assignment>
                           ! <compile-time if>
```

12.2 COMPILE-TIME STATEMENTS AND DECLARATIONS

12.2.1 COMPILE-TIME VARIABLES

Compile-time variables of type boolean may be declared by means of the compile-time declaration statement.

```
<compile-time declaration> ::=
     ?  var <compile-time var spec>
       {,<compile-time-var spec>} ?;
<compile-time var spec> ::=
   <identifier list> : <compile-time type> :=
   <compile-time expression>
<compile-time type> ::= boolean
```

The following rules apply:

12.0 COMPILE-TIME FACILITIES
12.2.1 COMPILE-TIME VARIABLES

1. The compile-time declaration statement must appear before the use of any of the compile-time variables. The scope of the compile-time variable is from the point of declaration to the end of the module.

2. Compile-time variables may be used only within compile-time expressions and compile-time assignment statements.

3. Identifiers of compile-time variables may not be the same as any other program identifiers.

12.2.2 COMPILE TIME EXPRESSIONS

Compile-time expressions must be composed only of constants and compile-time variables, but excluding identifiers for user-defined constants.

The operators defined on compile-time variables are:

and or xor not for type **boolean**

```
<compile-time expression> ::= <compile-time term>
     !<compile-time expression><disjunctive operator>
                                <compile-time term>

<compile-time term> ::= <compile-time factor>
     !<compile-time term> and <compile-time factor>

<compile-time factor> ::= true!false!<compile-time variable>
     !(<compile-time expression>)! not <compile-time factor>

<disjunctive operator> ::= or ! xor
```

12.2.3 COMPILE-TIME ASSIGNMENT STATEMENT

The value of a compile-time variable may be altered by a compile-time assignment statement.

```
<compile-time assignment> ::= ? <variable> :=
   <compile-time expression> ?;
```

12.2.4 COMPILE-TIME IF STATEMENT

The compile-time if statement is used to make the compilation of a piece of source code conditional upon the value of some boolean expression.

```
<compile-time if> ::=
     ? if <compile-time expression> then <text>
     [? else <text>]
     ? ifend
```

The following rules apply:

1) The expression must be a compile-time boolean expression.

2) Compilation of the <text> occurs only if the value is true.

Example:

```
   ? var small_size : boolean := true?;
   var Table : array [1..50] of integer ;
   ? if small_size = true then
      {might include this procedure call into program.}
      Bubblesort (Table);
   ? else
      {or call on procedure Quicksort in program.}
      Quicksort (Table);
   ? ifend
```

12.3 PRAGMAIS

Pragmats are used to specify and control:

A) Source and object text listings produced as by-products of compilation, and their layouts;

B) Layout aspects of the source text;

C) Kinds of run-time error checking;

D) Other aspects of the compilation process.

12.0 COMPILE-TIME FACILITIES
12.3 PRAGMATS

```
<pragmat statement> ::=
          ??  <pragmat> { ,<pragmat> } ??

<pragmat> ::= <toggle control>
            ! <layout control>
            ! <maintenance control>
            ! <comment control>
```

12.3.1 TOGGLE CONTROL

Uniquely identified control elements, called toggles, are used to control aspects of compilation. Each toggle is associated with a specific type of listing, run-time checking, or other activity. Toggles take on the value on or off. If on, the activity associated with the toggle is carried out, otherwise, it is not.

Toggle controls are used to:

A) Set the values of individual toggles;
B) Save and restore all toggle values in a LIFO manner;
C) Reset all toggles to their initial values.

(The initial settings of toggles are specified below.)

```
<toggle control> ::= set (<toggle setting list>)
                   ! push (<toggle setting list>)
                   ! pop
                   ! reset

<toggle setting list> ::= <toggle setting> {,<toggle setting>}
<toggle setting> ::= <toggle identifiers> := <condition>
                   ! <empty>

<condition> ::= on ! off
```

The operations are as follows.

Set: All settings specified in the list are carried out en-masse. If a toggle is affected by more than one toggle setting, the rightmost setting for that toggle is carried out.

Push: A record of the current state of all toggles is saved for future restoration in a LIFO manner; the current state remains intact. A set operation is then carried out.

Pop: The last state record saved becomes the current state. If none have been saved, the initial state becomes current.

12.0 COMPILE-TIME FACILITIES
12.3.1 TOGGLE CONTROL

Reset: The initial state becomes current, and any saved state records are wiped out.

The maximum allowable number of saved state records will be implementation dependent, but should not be less than one.

12.3.2 TOGGLES

<toggle identifiers> ::= <listing toggles>
| <checking toggles>

Toggle identifiers may be used freely for other purposes outside of pragmats.

12.3.2.1 Listing_Toggles

<listing toggles> ::= list | listobj
| listcts | listext | listall

List (initially is on): Controls all other listing toggles. When on, a source listing is produced, and other listing aspects are controlled by the other listing toggles. When off no listings can be produced.

Listobj (initially is off): Controls the listing of generated object code, which is interspersed with source code, following the corresponding source line.

Listcts (initially is off): Controls the listing of the format control pragmats. The format control pragmats are the listing toggles and the layout controls.

Listext (initially is off): When set to on the listing of source statements is externally controlled via a compiler call list option.

Listall: The union of all listing toggles. When set to on or off then all other listing toggles are set to on or off respectively.

12.3.2.2 Run-Time_Checking_Toggles

```
<checking toggles> ::= chkrng
                     ! chksub
                     ! chknil
                     ! chktag
                     ! chkall
```

Chkrng (default is on): controls the generation of object code that performs the range checking of scalar subrange assignments and that performs the range checking of case variables.

Chksub (default is on): controls the generation of object code that checks array subscripts (indices) and substring selectors to verify that they are valid.

Chknil (default is off): controls the generation of object code that checks for a nil value when a pointer dereference is made.

Chktag (default is on): controls the generation of object code that verifies that references to a field of a variant record are consistent with the value of its tag field (if a tag field is present).

Chkall: The union of all checking toggles; sets all four of chkrng, chksub, chknil, and chktag as a group.

The effects on the object code that is generated by these toggles being turned on or off is implementation and system dependent.

12.3.3 LAYOUT CONTROL

Layout controls are used to specify source text margins and to specify and control listing layout.

```
<layout control> ::= <source layout>
                   ! <listing layout>
```

12.3.3.1 Source_Layout

```
<source layout> ::= <source margin control>

<source margin control> ::=  left  := <left>
                           ! right := <right>
```

12.0 COMPILE-TIME FACILITIES
12.3.3.1 Source Layout

```
<left> ::= <integer>
<right> ::= <integer>

     {where 0 < left, and (left +10) <= right <= 110}
```

All source text to the left of the left-th and the right of the right-th position are ignored. Default values for left and right are 1 and 79 respectively.

12.3.3.2 Listing_Layout

```
<listing layout> ::= <pagination>
                   ! <lineation>
                   ! <titling>
```

12.3.3.2.1 PAGINATION

```
<pagination> ::= pagesize  := <pagesize>
               ! eject

<pagesize> ::= <integer>
               {20<= pagesize, default=58}
```

The pagesize value specified gives the maximum number of line positions that constitute a page. The first line position is called the top of page, and the last line position, the bottom.

The eject pragmat causes the paper to be advanced to the top of the next page.

12.3.3.2.2 LINEATION

```
<lineation> ::= spacing  := <spacing>
              ! skip  := <number of lines>

<spacing> ::= 1 ! 2 ! 3

<number of lines> ::= <integer>
                      {where 1 <= number of lines}
```

The spacing control may have the value 1, 2, or 3, for single, double, or triple spacing respectively. The default value is 1. A value of zero may not be used to indicate overprinting. Use of illegal values will result in no change in spacing, and an error message will be given.

The skip value causes a skip of the number of line positions specified; if the integer given is larger than pagesize or would cause a skip past the bottom of the current page, then the skip is to

the top of the next page.

12.3.3.2.3 TITLING

A standard title line is printed atop each page, and then one line position is skipped. Any additional titles defined by the user are then printed one-per-line, single-spaced. A skip of <spacing> number of lines then occurs.

<titling> ::=
    newtitle := '<char token> {<char token>}'
  ! title := '<char token> {<char token>}'
  ! oldtitle

A single quote mark within a char string is indicated by using a pair of adjacent single quote marks. Thus, if the char string were to consist of only a single quote mark, it would be indicated by four (4) immediately adjacent single quotes, e.g., ''''.

Newtitle: The current title is saved and the character string given as a new title becomes the current title. A standard page header is the first title printed on a page, followed by user-specified titles in the order in which they were saved; i.e., titles are saved in a LIFO manner, but are printed in a FIFO manner. There will always be a single empty line between the standard page header and the titles defined by the user. There will always be at least one blank line between the titles and the text or the standard header and the text.

The maximum number of titles allowed will be 10. An attempt to add more than the maximum will be ignored, without comment.

Title: The character string replaces the current user-defined title. If there is none, then the character string becomes the current title.

Oldtitle: The last user-defined title saved becomes the current title; is there is none, then no action is taken.

The titling does not take effect until the top of the next printed page.

12.3.4 MAINTENANCE CONTROL

<maintenance control> ::= compile ! nocompile

In the absence of a maintenance control, compile is the default option. The nocompile option continues with listing the following

text according to the listing toggles and layout controls, interpreting and obeying pragmat directives in the text, but compilation of the source is omitted until a **compile** directive is encountered or until a **modend** statement is encountered.

12.3.5 COMMENT CONTROL

<char token> ::= **comment** := '<char token>[<char token>]'

Including the comment control pragmat signals the compiler to include the character string in the binary output generated by the compilation process. This allows for COPYRIGHTing products and for commenting object code facilities like load maps.

13.0 IMPLEMENTATION-DEPENDENT FEATURES

13.0 IMPLEMENTATION-DEPENDENT FEATURES

In contrast to the previously discussed aspects of the language, the language features discussed in this section may be dependent upon the compiler's allocation algorithms or the hardware design. These features may be used anywhere, but should be used with caution.

13.1 DATA MAPPINGS

The mapping of data storage will depend on a compiler's target machine and data mapping algorithms. All effects of data mapping will, therefore be implementation dependent: bit-sizing, positioning, relative positioning effects of packing attributes. Data mapping algorithms for specific implementations may be published; these can be used to achieve specific sizings and positionings for that implementation.

APPENDIX A = CYBIL METALANGUAGE CROSS-REFERENCE

APPENDIX B - CYBIL RESERVED WORD LIST