LANGUAGE SPECIFICATION for the CYBER IMPLEMENTATION LANGUAGE (CYBIL)

Written By: ________________
H.A.Wohlwend

Approved By: ________________

____

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.
7	12/11/81	Updated to reflect ILDT approved language changes and miscellaneous clarifications.
8	03/06/86	Updated to reflect ILDT approved language changes and miscellaneous clarifications.

Table of Contents

1.0 INTRODUCTION

The CYBER Implementation Language (CYBIL) language is the implementation language for Control Data Corporation. This document provides the definition for the CYBIL language.

This specification was developed from Rev. 7 of this specification and from DAP's S4304, S4478, S4497, S4505, S4545, S4547 S4552, S4691, S4765, S4802, S4874, S4925, S4953, ARH5266, ARH5267 and ARH5268. These updates have Implementation Language Design Team approval and DCS review cycle approval.

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 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

3.1 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 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.

```
<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

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 Facilities, 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>
                |<cell type>
                |<pointer type>
                |<relative pointer 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>

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

$char(0) .. $char(255)

where "$char" 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. 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 $integer function. For the constants of the example, the following relations hold:

```
  $integer (red) = 0
  $integer (green) = 1
  $integer (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>
```

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, exclusive or, 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. The parent type of the subrange is the type of the lower and upper constant expression.

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),
      hot color = 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 Cell Type

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

<cell type identifier> ::= <identifier>
```

A cell type is a basic type that represents the smallest storage site that is directly addressable by a pointer. It is not equivalent with any other type.

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

4.3.1.4 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; any CYBIL type is legal. Pointer variables may assume, as values, only pointers to that type. The only exception to this is pointer to cell.

```
<pointer type> ::= <fixed pointer>
                  |<fixable pointer>
                  |<pointer to procedure>
                  |<pointer to function>
                  |<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 to function> ::= ↑<function 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, a push 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.

4.3.1.4.1 POINTER TO CELL

```
<pointer to cell> ::= ↑cell
```

A pointer to cell is a pointer type.

<u>Permissible Operations</u>: 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, adaptable or bound variant type.

4.3.1.5 Relative Pointer Types

Relative pointer types represent relative locations (with respect to the beginning of some composite object) of components of such objects.

```
<relative pointer type> ::=
     rel (<parental type>) ↑ <object type>

<parental type> ::= <storage type>
                  | <adaptable storage type>
                  | <aggregate type>
                  | <adaptable aggregate type>

<object type> ::= <type>
```

Relative pointers provide three facilities not given by pointer types:

1. A relative pointer variable may require less space than a pointer variable.

2. A linked list or array of relative pointers (or a similar pointer network) within a parental variable is still correct if that

entire variable is assigned to another variable of the same parental type.

3. Relative pointers are independent of the base address of the parental variable.

Relative pointer values can be generated solely through the built-in function <u>#rel</u> whose arguments are a pointer variable and an optional parental variable.

Relative pointers cannot be used to access data directly. Such data must be accessed through a pointer generated by the built-in function <u>#ptr</u> whose arguments are a relative pointer variable and an optional parental variable.

Relative pointer types are equivalent if they are defined in terms of equivalent parental types and equivalent object types.

<u>Permissible Operations</u>: assignment, #PTR function, and comparison for equality and inequality only. Relative pointers are assignable and comparable if they are of equivalent relative pointer types.

4.3.2 STRUCTURED TYPES

Structured types represent collections of components, and are defined by describing their component types and indicating a so-called <u>structuring</u> <u>method</u>. 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, heaps, or sequences 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>
```

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

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

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

With a <selection spec> of the form constant scalar expression1 .. constant scalar expression2 the following rule applies: lowervalue (<tag field type>) <= <constant scalar expression1> <= <constant scalar expression2> <= uppervalue (<tag field type>). The subrange selection specification signifies all of the constants in the inclusive range from constant scalar expression1 up through and including constant scalar expression2. It is semantically equivalent to having all the constants in the range, constant scalar expression1 through constant scalar expression2, listed separately in selection specs.

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 the same packing, their fixed parts, considered as invariant record types, are equivalent, their tag field selectors are identical, their tag field types are equivalent, their selection specs are the same, and their corresponding variants, considered as record types (either variant or invariant) are equivalent. Note that this definition is recursive.

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,
   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>)
                  | <sequence type identifier>
```

```
<sequence type identifier> ::= <identifier>
```

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>)
              | <heap type identifier>

<heap type identifier> ::= <identifier>
```

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 default 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
                                | real | longreal | 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 aggregate 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, equivalent component types and indexes of integer 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.

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 FUNCTION TYPE

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

A function type defines an optional ordered list of formal parameters together with a return type.

```
<function type> ::= <function type identifier>
```

```
                    |function <func type spec>

<function type identifier> ::= <identifier>
```

Function types are used for declaration of pointers to functions, there are no function variables. A "pointer to function" by default will be unsafe.

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

4.7 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. It is not legal to assign directly into the tag field selector for a bound variant record.

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.

4.8 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.

4.9 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. For a field of a packed record to be passed as a reference parameter the field must be aligned. Aligning the first field of a record aligns the record.

A second usage of the alignment feature is to cause variables of type record, 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 a machine dependent value, reflecting the characteristics of the machine addressing mechanisms. 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 a machine dependent value, reflecting the characteristics of the machine addressing mechanisms.

The <offset> and <base> elements are cell counts.

4.10 OTHER ASPECTS OF TYPES

4.10.1 VALUE AND NON-VALUE TYPES

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

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

4.10.2 COMPARABLE AND NON-COMPARABLE TYPES

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

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

4.10.3 FUNCTION-RETURN TYPES

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

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

4.10.4 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 <u>$integer</u> function.

B) Integer values between 0 and 255 are convertible to characters by the <u>$char</u> function.

4.10.5 TYPE MIXING

Any variant record whose purpose is to allow type casting (conversion) of one given data structure onto another must only modify the variants directly; the use of pointer indirection to change such a record variant may cause undefined results. The CYBIL language and supporting compilers guarantee support only for this immediate type casting; indirect type casting violates language rules and is not supported.

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>

<character constant> ::= '<char token>'
                       |$char (<integer constant>)
                       |<character constant identifier>

<char token> ::= <alphabet>
               | '' {two apostrophes}

<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>

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

<radix> ::= 2 | 8 | 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, constant identifiers or parenthesized constant expressions.

B) The expressions must be simple expressions (terms involving relationals must be parenthesized).

C) The only functions allowed as factors in such expressions are the _$integer_, _$char_, _succ_ and _pred_ functions with constant expressions as arguments.

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.

5.2 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 constant 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

6.1 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 or function 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.

means: storage goes away autom. on exit from proc.

6.2.2.3 Lifetime Conventions

If neither storage attributes nor scope attributes are specified, then the variable is treated as an automatic variable, unless the variable is at the outermost level of a module body.

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.

[illegible]

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.

If the variable is a formal parameter, then the pointer cannot be used to modify the parameter.

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}

   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.

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 $integer;

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>}

<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

7.1 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>
       modend [<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. 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 immediately containing block.

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 discontinuous 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 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

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 attributes>]]<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 is declared, and remains fixed throughout the execution of that block.

8.0 PROCEDURES AND FUNCTIONS
8.1 PROCEDURE DECLARATIONS

```
<proc attributes> ::= <proc attribute> , {<proc attribute>}

<proc attribute> ::= xdcl | inline | #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>

<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 #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.

The inline attribute directs the compiler to substitute the procedure statement body at the point of call to the procedure rather than actually calling the procedure. Certain restrictions may exist for the inline procedure candidates.

8.2 FUNCTION DECLARATIONS

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

   <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> | unsafe

<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.

There are two kinds of function declarations provided for in the CYBIL language. One provides for functional notation where there can be no undesirable side effects and the other provides for functional notation in a form where side effects are possible.

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 "safe" functions.

The left-hand side of an assignment statement within a function may not be any of the following: e.g. XDCL variable or local [illegible] able of outer nesting,

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

User defined "safe" functions may not contain:

- o Procedure call statements that call user-defined procedures,

- References to unsafe functions,
- Parameters of type pointer to procedure or pointer to function,
- ALLOCATE, FREE, PUSH, RESET or NEXT statements that have parameters that are not local variables.

These restrictions may make it necessary to use an unsafe function or a procedure for some purposes for which a "safe" 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 INLINE PROCEDURES AND FUNCTIONS

The following considerations apply for inline procedures and functions:

- Type, constant and variable declarations local to an inline procedure or function are appended to the declarations for the calling procedure or function. These types, etc. may be referenced only in the inline procedure or function body as all the normal naming and scoping rules for identifier definition and referencing still apply.
- Local (non-XREF) static variable definitions are not permitted.
- An inline procedure or function may not contain nested procedure or declarations, except for XREF'ed procedures.
- An inline procedure or function may reference any other procedure or function, including other inline procedures or functions. Recursive calls to an inline procedure or function, either directly or indirectly, are not allowed.
- Space allocated by a PUSH statement in an inline procedure or function is not de-allocated until the calling (non-inline) procedure or function exits.
- The identifier for an inline procedure or function may not be used in a pointer reference.

8.5 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.

8.6 EXAMPLES OF PROCEDURES AND FUNCTIONS

```
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

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

<constant identifier> ::= <identifier>

<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>)
                    |$char (<expression>)
                    |$integer (<expression>)
                    |$real (<expression>)
                    |$longreal (<expression>)
                    |strlength (<fixed string type identifier>
                               |<string variable>)
                               |<string constant>)
                               |<string constant identifier>)
                    |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>)
                    |#rel (<pointer>[,<parental>])
                    |#ptr (<relative pointer>[,<parental>])
                    |#seq (<variable reference>)
                    |#loc (<variable>)
                    |#size(<variable>
                         |<fixed type identifier>
                         |<adaptable type> : [<adaptable field fixer>])

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

<string constant identifier> ::= <identifier>

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

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

<scalar variable> ::= <variable>

<parental> ::= <parental type variable>

<parental type variable> ::= <variable>
```

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

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 hue1
```

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 apostrophe marks.

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

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

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.

9.2.2 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 real longreal	integer real 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 $\equiv$ + a).

The - operator can be applied to integer, real, longreal and set types only. It denotes sign inversion--i.e., -a $\equiv$ 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.

Operator	Operations	Operands	Result
+	addition	integer or integer subrange real longreal	integer real 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 real longreal	integer real 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 contained in either set but not both sets.	set of type T	set of type T

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 $integer(a) op $integer(b) holds, where op denotes any of the six comparison operators and $integer 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 Relative Pointers

Relative pointer comparison is allowed only for relative pointers of equivalent type. Two relative pointers are equal if the relationship #ptr (p,P) = #ptr (q,P) holds, where p and q denote relative pointers of equivalent type, and P denotes a variable whose type is equivalent to the parental types of these relative pointers.

A relative pointer of any type may be compared for equality or inequality with the value nil. A relative pointer comparison results in equality if both relative pointers have the value nil.

9.2.5.4 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.

9.2.5.5 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≤k≤n
        a(k) op b(k) AND
        a(i) = b(i) for 1≤i<k
```

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

o a < = b iff a = b OR a < b

Comparing two null strings results in equality.

9.2.5.6 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.

9.2.5.7 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.

9.2.5.8 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.

9.2.5.9 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)	string(n)	boolean
		S(k) *	char	boolean
		char	S(k) *	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

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:

- o Any variable specified as read-only, or a formal value parameter of any containing procedure.
- o Any bound variant record.
- o The tag field of any bound variant record.
- o 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:

- o The types of the variable and the expression may be subranges of equivalent parent types. If the value of the expression is not a value of the type of the variable, the program is in error.
- o 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.
- o 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.
- o 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 lifetime must not survive the lifetime of the data to which it is pointing. For example, a static pointer variable cannot point to a local variable. This rule also applies to a pointer assigned by an allocation statement.

- 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 or a pointer to sequence, 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 pointer type other than pointer to sequence, 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 or function declaration, then its scope is that procedure or function declaration. It is impossible to refer to a structured statement designator on a structured statement from outside that statement. A structured statement designator may optionally follow a structured statement (except repeat.. until), in which case it must be identical to the structured statement designator 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 designator> ::=
             / <structured statement identifier> /
<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 designator>]
                            <repeat statement>
          |[<structured statement designator>] <delimited statement>
                            [<structured statement designator>]

<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.

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>
```

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.

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>})
     | <empty>

<empty> ::=

<actual parameter> ::= <expression>
                     | <variable>
                     | <empty>
```

The actual parameter list must be compatible with the formal parameter list of the procedure. An actual parameter corresponds to the formal parameter which occupies the same relative position in the formal parameter list.

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. In the form constant scalar expression1 .. constant scalar expression2 the value of constant scalar expression1 must be less than or equal to the value of constant scalar expression2. It signifies all of the constants in the inclusive range from constant scalar expression1 up through and including constant scalar expression2. It is semantically equivalent to having all the constants in the range constant scalar expression1 through constant scalar expression2 listed separately in selection specs.

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

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=
      rvalue : real,
    =stringconst=
      length : 1..255,
      stringbuf : ↑string(* <= 255),
    =identifier=
      identno : integer,
      decl : ↑symbolentry,
    casend,
  recend;

var
  cursym : symbol,
  sign : [static] 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.rvalue := -cursym.rvalue;
      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 := table[1];
  /find_smallest/
    for k := 2 to n do
      if x < table[k] then
        cycle /find_smallest/;
      ifend;
      x := table[k]; {this assignment skipped when x < table[k]}
       {this finds the smallest value in table[1] thru table[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 <exit designator>

<exit designator> ::= <structured statement designator>
                    | <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 designator is 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 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 procedure or function identifier. In the case of recursive nests, the result is exiting the most recent invocation of the procedure or function specified, and any intervening procedures of functions which have been activated.

Examples:

```
/meaningful_label/
 begin                        {example of exit <label>}
    x := y + 27;
    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 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>
```

```
<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> ::= <non-negative integer expression>
<span fixer> ::= [<span> {, <span> }]
<span> ::= [rep <non-negative 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.

   The lower bound is permitted to exceed the upper bound by one. In this case a valid address is assigned to the adaptable array pointer variable, but no storage is allocated. The adaptable array pointer variable is set to designate an array with the specified upper and lower bounds.

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. of record

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.

[illegible] when array

6) For allocation designators for adaptables, entries are required only for the dimension 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 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 The Stack

A variable allocated on the 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
    {the 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 when the data is retrieved from a sequence must be equivalent to the pointer variable as when that same data was stored into the sequence; otherwise, the program is in error.

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

<pointer to sequence reference> ::= <pointer to sequence variable>
                                  | <function reference>

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

The operation of the next statement is defined in terms of two cursors: the present_cursor and the next_cursor. For the next operation, the present_cursor is set to the next_cursor, the next_cursor is incremented by the size of the type of the allocation designator, and the variable is set to the location value of the present_cursor.

If the execution of a next statement would cause the new next_cursor to lie outside the bounds of the sequence, then the allocation designator is set to the value nil and the cursor positions remain unchanged.

present := next
[illegible]
to the 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 reference>]
   | reset <heap variable>
```

Warning: a res
statement for any
or heap to an 'em

10.6.4.1 Reset Se

The reset seq
contained in a
the optional to
sequence become
specified, the e
variable> becom

Note: present-cursor & next-cursor
are needed because:
present-curser points to address
of element nexted (alloc. in Sequ.)
When another element is nexted the
sequence mgt. must know the
address of next free location
(i.e. must know length of last
nexted element)

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 User 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 default heap.

Note that the first allocate statement for any heap (other than the default 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 reference>
```

Examples:

```
   var my_array : ↑array [0 ..  *] of integer;

   allocate my_array : [0..49]; {allocate space in default 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 default 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.

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

Examples:

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

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 {,P})

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

The string representation procedure facilitates the conversion of <concatenation element>s to their representation as a string of characters.

One or more <concatenation element>s are converted into output fields consisting of strings of characters. The resulting output fields are concatenated and 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 string representation of the resulting string exceeds the length of the <string variable> S, then right truncation occurs and the <result length> L becomes the length of the <string variable> S.

11.1.1.1 Concatenation Elements

```
<concatenation element> ::= <scalar element>
                          | <string element>
                          | <pointer element>
                          | <floating point element>

<scalar element> ::=
     <scalar expression>[<scalar field specifier>]

<scalar field specifier> ::=
          [:<field length>] [:<radix spec>]

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

<radix spec> ::= #(<radix>)

<string element> ::=
     <string expression> [<string field specifier>]

<string expression> ::= <string variable>
                      | <string constant>
                      | <substring reference>

<string field specifier> ::= :<field length>

<pointer element> ::=
          <pointer reference>[<pointer field specifier>]

<pointer field specifier> ::= [:<field length>][:<radix spec>]

<floating point element> ::=
   <floating point expression> [<floating point field specifier>]

<floating point expression> ::= <real expression>
                              | <longreal expression>

<real expression> ::= <expression>

<longreal expression> ::= <expression>

<floating point field specifier> ::=
          : <field length> [:<fractional digits>]

<fractional digits> ::= <positive integer expression>
```

In general, numeric values are written right justified into the specified field, with blank left fill or filled with asterisk (*) characters if truncation would have occurred. Values specified to be in string or character (alphabetic) form are written left justified into the specified field, with blank right fill or filled with asterisk (*) characters if truncation would have occurred. In all cases, the value of the field length, when specified, must be greater than or equal to zero or an error will occur.

11.1.1.1.1 INTEGER ELEMENT

The value of the integer expression is converted into a string representation in the desired radix. The default radix value is 10. The resulting string representation is placed right justified into the output field with leading blanks if a field length greater than required was specified. If the field length given is not long enough to contain all the digits and the sign character of the value of the integer expression, then the output field is filled with a string of asterisk characters. 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 length is omitted, then the output field is the minimum size required to contain the integer value plus the necessary leading character. If the field length specified is less than or equal to zero an error will occur.

11.1.1.1.2 ORDINAL ELEMENT

The integer value of the ordinal expression is handled in exactly the same manner as an integer element.

11.1.1.1.3 SUBRANGE ELEMENT

A concatenation element which is a subrange type is handled exactly as the type of which it is a subrange.

11.1.1.1.4 CHARACTER ELEMENT

The single string character is placed <u>left</u> justified into the output field with trailing blanks if a field length greater than required was specified. The default field length is 1. Quoting the radix spec for character elements is a compilation error.

11.1.1.1.5 BOOLEAN ELEMENT

The five character string ' TRUE' or 'FALSE' is placed <u>left</u> justified into the output field with trailing blanks if a field length greater than required was specified. 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. The default field length is 5. Quoting the radix spec for boolean elements is a compilation error.

11.1.1.1.6 STRING ELEMENT

The string expression is placed left justified into the output field with trailing blanks if a field length greater than required was specified. If the field length given is shorter than the length of the string, then the output field is filled with a string of asterisk characters. If the field length is omitted, then the output field is the minimum size required to contain the string expression.

11.1.1.1.7 POINTER ELEMENT

The value of the pointer expression is converted into a string representation in the desired radix. The default radix value is implementation dependent and will depend on the characteristics of the native machine. The resulting string representation depends on the type of pointer involved, and is system and machine dependent.

The resulting string representation is placed right justified into the output field with leading blanks if a field length greater than required was specified. If the field length given is not long enough to contain all the digits, then the output field is filled with a string of asterisk characters. If the field length is omitted, then the output field is the minimum size required to contain the pointer value.

11.1.1.1.8 FLOATING POINT ELEMENT

A floating point expression can be converted into either a fixed point format or a floating point format depending on the <floating point field specifier>. If there is no <floating point field specifier> then the conversion is done as if <field length> had been specified with an implementation defined value.

11.1.1.1.8.1 Floating Point Format

E:field_length will cause conversion into an output string of length field_length. It will contain a mantissa/exponent representation of E with at most max_real_digits or max_longreal_digits, which are implementation defined, in the mantissa. The exponent will contain num_exp_digits which is implementation defined. Let Exponent be the integer such that

```
10**Exponent <= ABS ( E ) < 10**(Exponent+1)
If E is real then
 num_digits := MIN(field_length-4-num_exp_digits, max_real_digits)
If E is longreal then
 num_digits := MIN ( field_length - 4 - num_exp_digits,
 max_longreal_digits)
```

If num_digits is less than 1 then the output field will be filled with asterisks. Otherwise, the output field will consist of:

1) if field_length > num_digits +4 +num_exp_digits then (field_length -num_digits -4 -num_exp_digits) spaces

2) if E < 0 then '-' else one space

3) the leading digit of the decimal representation of E after rounding to num_digits places

4) the character '.'

5) the next (num_digits-1) digits of the decimal representation of E after rounding to num_digits places

6) the character 'E' for real expressions or 'D' for double precision expressions

7) '+' or '-' depending on the sign of Exponent

8) num_exp_digits representing Exponent with '0' fill on the left if needed.

```
Examples: format    E          output string
          E:10      123.456    ' 1.23E+002'
          E:11     -123.456    '-1.235E+002'
```

11.1.1.1.8.2 Fixed Point Format

E:field_length:fractional_digits will cause the expression E to be converted to an output string of length field_length with fractional_digits to the right of the decimal place. If fractional_digits is less than zero or greater than (field_length-2) then the program is in error. A size error will be generated if checking is enabled. Let E_out be the decimal representation of E rounded to have fractional_digits to the right of the decimal point and one zero to the left of the point if TRUNC(E_out)=0. Let num_left_digits be the number of digits to the left of the decimal point in E_out.

```
required_length : = num_left_digits +1 +fractional_digits;
if E_out < 0 then
required_length := required_length + 1; {'-' required}
```

If field_length < required_length then the output string will consist of all asterisks. Otherwise, it will consist of:

1) if field_length > required_length then

(field_length-required_length) spaces

2) if E_out < 0 then '-' else one space

3) the first num_left_digits of E_out

4) the character '.'

5) the fractional_digits of E_out to the right of the decimal point.

```
Example:      format    E         output string
              E:6:2     1.23456   '  1.23'
              E:6:3    -1.23456   '-1.235'
              E:5:2     0         ' 0.00'
```

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 $CHAR(X)

Returns the character value whose ordinal number, in the ASCII collating sequence, is given by the integer expression, X. If the value of X lies outside that range (0 <= X <= 255), an out-of-range error occurs.

11.2.4 $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.

11.2.5 $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.

11.2.6 $LONGREAL(X)

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

11.2.7 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, a string type identifier, a string constant or a string constant identifier. For an adaptable string this is the current length and x must be an adaptable string reference.

11.2.8 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.

11.2.9 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.

11.2.10 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.

11.2.11 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.

11.2.12 #REL (POINTER[,PARENTAL])

This function produces a relative pointer value from a pointer variable and parental variable. If the parental variable is not supplied, the default heap is used. The relative pointer's object type is the object type of the pointer variable, and its parental type is that of the parental variable. The result is undefined if the pointer does not designate an element of the parental variable.

11.2.13 #PTR (RELATIVE POINTER[,PARENTAL])

This function is used to convert a relative pointer to a pointer, and is required when using a relative pointer to access the object pointed to by the relative pointer. It returns a pointer to the same type as the object type of the relative pointer. If the parental variable is not specified then the default heap is used. If the parental type associated with the relative pointer is not equivalent to the type of the parental variable, an error results.

11.2.14 #SEQ (VARIABLE)

Returns a pointer to sequence that designates the argument variable. The argument variable may be of any type. The following relations hold:

```
        #LOC( #SEQ( x )↑ = #LOC( x )
        #SIZE( #SEQ( x )↑ = #SIZE( x )
```

11.3 REPRESENTATION DEPENDENT FUNCTIONS

11.3.1 #LOC(<VARIABLE>)

Returns a pointer to the first cell allocated for the specified variable. If the variable is a formal parameter, then the pointer cannot be used to modify the parameter.

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.

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:

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 PRAGMATS

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) Object libraries associated with this compilation unit;

E) Other aspects of the compilation process.

```
<pragmat statement> ::=
          ??  <pragmat> { ,<pragmat> } ??

<pragmat> ::= <toggle control>
            | <layout control>
            | <maintenance control>
            | <comment control>
            | <object library 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 last in-first out 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 last in-first out 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.

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 lines 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>

<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> ::= eject
```

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
```

An apostrophe mark within a char string is indicated by using a pair of adjacent apostrophe marks. Thus, if the char string were to consist of only an apostrophe mark, it would be indicated by four (4) immediately adjacent apostrophes, 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 last in-first out manner, but are printed in a first in-first out 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

<comment control> ::= 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.

12.3.6 OBJECT LIBRARY CONTROL

<object library control> ::= library := <library name>

<library name> ::= '<alphabet> {<alphabet>}'

Including the object library control pragmat signals the compiler to include the library name in the library directive of the binary output produced during the compilation process. This allows linking the xref declarations with the appropriate object library.

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

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


   1     10-10 <a case> ::= =<selection spec>{,<selection spec>}=
                              <statement list>

   2      6-3  <access attribute> ::= read

   3     10-7  <actual parameter> ::= <expression>
                                    | <variable>
                                    | <empty>

   4     10-7  <actual parameter list> ::=
                      (<actual parameter>{,<actual parameter>})
                     | <empty>

   5     4-19  <adaptable aggregate type> ::= <adaptable string>
                                             |<adaptable array>
                                             |<adaptable record>

   6     4-20  <adaptable array> ::=
                      [packed]<adaptable array identifier>
                     | [packed]<adaptable array spec>

   7     4-20  <adaptable array bound spec> ::= <lower bound spec> ..   *
                                               | *

   8     4-20  <adaptable array identifier> ::= <identifier>

   9     10-15 <adaptable array pointer variable> ::= <pointer variable>

  10     4-20  <adaptable array spec> ::=
                     array [<adaptable array bound spec>] of <component type>

  11     4-21  <adaptable field> ::=
                  <field selector>:[<alignment>]<adaptable type>

  12     10-16 <adaptable field fixer> ::= <star fixer>
                                          |<length fixer>
                                          |<span fixer>

  13     4-20  <adaptable fixed string> ::= string (<adaptable string length>)

  14     4-22  <adaptable heap> ::= heap(*)
                                   |<adaptable heap identifier>

  15     4-22  <adaptable heap identifier> ::= <identifier>
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION

  16      4-8  <adaptable pointer> ::= ↑<adaptable type>

  17     4-21  <adaptable record> ::=
                     [packed]<adaptable record type identifier>
                    |[packed]<adaptable record spec>

  18    10-16  <adaptable record pointer variable> ::= <pointer variable>

  19     4-21  <adaptable record spec> ::=
                  record[<fixed fields>,]<adaptable field><recend>

  20     4-21  <adaptable record type identifier> ::= <identifier>

  21     4-21  <adaptable sequence> ::= seq (*)
                                    |<adaptable sequence identifier>

  22     4-21  <adaptable sequence identifier> ::= <identifier>

  23    10-16  <adaptable storage pointer variable> ::= <pointer variable>

  24     4-19  <adaptable storage type> ::= <adaptable sequence>
                                            <adaptable heap>

  25     4-20  <adaptable string> ::= <adaptable fixed string>
                                    | <adaptable string identifier>

  26     4-20  <adaptable string bound> ::= <length>

  27     4-20  <adaptable string identifier> ::= <identifier>

  28     4-20  <adaptable string length> ::= * | * <= <adaptable string bound>

  29    10-16  <adaptable string pointer variable> ::= <pointer variable>

  30     4-19  <adaptable type> ::= <adaptable aggregate type>
                                  |<adaptable storage type>

  31      9-1  <adding operator> ::= + | - | or | xor

  32     4-10  <aggregate type> ::= <string type>
                                  |<array type>
                                  |<record type>

  33      7-4  <alias> ::= alias ' <alphabet> { <alphabet> } '
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  34     4-24  <alignment> ::= aligned [[<offset> mod <base>]]

  35    10-20  <allocate statement> ::= allocate <allocation designator>
                         [ in <heap variable>]


  36    10-15  <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>]

  37      3-3  <alphabet> ::= <letter>
                             |<digit>
                             |<special mark>
                             |<blanks>
                             |<unused mark>

  38     4-13  <array spec> ::=
                      array [<index>] of <component type>

  39     4-13  <array type> ::= [packed]<array type identifier>
                               |[packed]<array spec>

  40     4-13  <array type identifier> ::= <identifier>

  41     6-13  <array variable> ::= <variable>

  42      3-3  <ascii character> ::= <alphabet>
                                    |<unprintable>
                                    |<string delimiter>

  43     10-2  <assignment statement> ::= <variable> := <expression>

  44      6-3  <attribute> ::= <access attribute>
                              |<storage attribute>
                              |<scope attribute>

  45      6-3  <attributes> ::= [<attribute>{,<attribute>}]

  46     4-24  <base> ::= <integer constant>
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  47      5-2  <base designator> ::= (<radix>)


  48     4-11  <base type> ::= <scalar type>

  49      5-1  <basic constant> ::= <scalar constant>
                                   |<floating point constant>
                                   |<pointer constant>

  50      4-3  <basic type> ::= <scalar type>
                               |<floating point type>
                               |<cell type>
                               |<pointer type>
                               |<relative pointer type>

  51     10-5  <begin statement> ::=
                    begin <statement list> end

  52      3-3  <blanks> ::=

  53      5-1  <boolean constant> ::= false | true
                                    | <boolean constant identifier>

  54      5-1  <boolean constant identifier> ::= <identifier>

  55      4-5  <boolean type> ::= boolean
                                 |<boolean type identifier>

  56      4-5  <boolean type identifier> ::= <identifier>

  57      4-8  <bound variant pointer> ::= ↑<bound variant record type>

  58    10-16  <bound variant record pointer variable> ::= <pointer variable>

  59     4-23  <bound variant record type identifier> ::=
                 <variant record type identifier>

  60     4-23  <bound variant record type> ::=
                 [packed] <bound variant record type identifier>
                |[packed] bound <variant record spec>
                |[packed] bound <variant record type identifier>
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION

  61      9-2  <built-in function reference> ::= succ (<scalar expression>)
                         |pred (<scalar expression>)
                         |$char (<expression>)
                         |$integer (<expression>)
                         |$real (<expression>)
                         |$longreal (<expression>)
                         |strlength (<fixed string type identifier>
                                    |<string variable>)
                                    |<string constant>)
                                    |<string constant identifier>)
                         |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>)
                         |#rel (<pointer>[,<parental>])
                         |#ptr (<relative pointer>[,<parental>])
                         |#seq (<variable reference>)
                         |#loc (<variable>)
                         |#size(<variable>
                               |<fixed type identifier>

  62     4-15  <case part> ::= case <tag field spec> of
                                        <variations><casend>

  63     10-9  <case statement> ::= case <selector> of <cases>

  64     4-16  <casend> ::= [,] casend
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  65    10-10  <cases> ::= <a case>{;<a case>}

  66      4-7  <cell type> ::= cell
                             | <cell type identifier>

  67      4-7  <cell type identifier> ::= <identifier>

  68      5-1  <char token> ::= <alphabet>
                              | '' {two apostrophes}

  69      5-1  <character constant> ::= '<char token>'
                                       |$char (<integer constant>)
                                       |<character constant identifier>

  70      5-1  <character constant identifier> ::= <identifier>

  71      4-4  <character type> ::= char|<character type identifier>

  72      4-4  <character type identifier> ::= <identifier>

  73     12-5  <checking toggles> ::= chkrng
                                    | chksub
                                    | chknil
                                    | chktag
                                    | chkall

  74      3-5  <comment character> ::= <any ASCII character except
                                          a closing brace or end of line>

  75     12-8  <comment control> ::= comment := '<char token>[<char token>]'

  76      3-5  <comment terminator> ::= } | <end of line>

  77      3-5  <commentary string> ::= {{<comment character>}
                                       <comment terminator>

  78      7-1  <compilation unit> ::= <module declaration>
                                    {;<module declaration>} [;]
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  79     12-2  <compile-time assignment> ::= ?  <variable> :=
                  <compile-time expression> ?;

  80     12-1  <compile-time declaration> ::=
                    ?  var <compile-time var spec>
                      {,<compile-time var spec>} ?;

  81     12-2  <compile-time expression> ::= <compile-time term>
                    |<compile-time expression><disjunctive operator>
                                              <compile-time term>

  82     12-2  <compile-time factor> ::= true|false|<compile-time variable>
                    |(<compile-time expression>)| not <compile-time factor>

  83     12-2  <compile-time if> ::=
                    ?  if <compile-time expression> then <text>
                   [? else <text>]

  84     12-1  <compile-time statement> ::= <compile-time declaration>
                                           | <compile-time assignment>
                                           | <compile-time if>

  85     12-2  <compile-time term> ::= <compile-time factor>
                    |<compile-time term> and <compile-time factor>

  86     12-1  <compile-time type> ::= boolean

  87     12-1  <compile-time var spec> ::=
                  <identifier list> : <compile-time type> :=
                  <compile-time expression>

  88     4-13  <component type> ::= <fixed type>

  89     11-2  <concatenation element> ::= <scalar element>
                                 | <string element>
                                 | <pointer element>
                                 | <floating point element>

  90     12-4  <condition> ::= on | off

  91      5-1  <constant> ::= <basic constant>|<string constant>

  92      5-3  <constant declaration> ::=
                     const <constant spec> {, <constant spec>}
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION

  93      5-3  <constant expression> ::= <simple expression>

  94    10-16  <constant fixers> ::= <constant scalar expression>
                                 {,<constant scalar expression>}

  95      9-1  <constant identifier> ::= <identifier>

  96     4-21  <constant integer expression> ::= <constant expression>

  97      5-3  <constant scalar expression> ::= <constant expression>

  98      5-3  <constant spec> ::= <identifier> = <constant expression>

  99     10-7  <control statement> ::= <procedure call statement>
                          | <if statement> | <case statement>
                          | <cycle statement>
                          | <exit statement> | <return statement>
                          | <empty statement>

 100     10-6  <control variable> ::= <variable identifier>

 101    10-12  <cycle statement> ::= cycle <structured statement identifier>

 102      7-2  <declaration> ::= <type declaration>
                              | <constant declaration>
                              | <variable declaration>
                              | <procedure declaration>
                              | <function declaration>
                              | <section declaration>
                              | <empty>

 103      7-1  <declaration list> ::= {<declaration>;}

 104     10-4  <delimited statement> ::= <begin statement>
                          | <while statement>
                          | <for statement>

 105      3-3  <digit> ::= 0|1|2|3|4|5|6|7|8|9

 106     12-2  <disjunctive operator> ::= or | xor

 107     10-7  <empty> ::=

 108    10-14  <empty statement> ::=
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


 109    10-13  <exit designator> ::= <structured statement designator>
                                    | <procedure identifier>
                                    | <function identifier>

 110    10-13  <exit statement> ::= exit <exit designator>

 111      5-2  <exponent> ::= [<sign>]<digit>{<digit>}

 112      9-1  <expression> ::= <simple expression>
                       |<simple expression><relational operator>
                                    <simple expression>

 113      9-1  <factor> ::= <variable>|<constant>|<constant identifier>
                    |<set value constructor>|<function reference>
                    |↑<procedure identifier>|↑<variable>
                    |(<expression>) |not<factor>

 114     11-2  <field length> ::= <positive integer expression>

 115     6-15  <field reference> ::=
                  <variable reference>.<record subreference>{.<record subrefere

 116     4-14  <field selector> ::= <identifier>

 117     4-14  <field selectors> ::= <field selector> {,<field selector>}

 118     10-6  <final value> ::= <scalar expression>

 119     6-11  <first char> ::= <positive integer expression>

 120      4-8  <fixable pointer> ::= <adaptable pointer>
                                   |<bound variant pointer>

 121      4-1  <fixable type> ::= <adaptable type>
                                |<bound variant record type>

 122      9-2  <fixed array type identifier> ::= <array type identifier>


 123     4-14  <fixed field> ::= <field selectors> : [<alignment>] <fixed type>

 124     4-14  <fixed fields> ::= <fixed field> {, <fixed field>}

 125      4-8  <fixed pointer> ::= ↑<fixed type>
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  126   10-15  <fixed pointer variable> ::= <pointer variable>

  127    4-18  <fixed span> ::=
                    [rep <positive integer constant expression> of]
                         <fixed type identifier>

  128    4-12  <fixed string> ::= string (<length>)

  129     9-2  <fixed string type identifier> ::= <string type identifier>


  130     4-3  <fixed type> ::= <basic type>|<structured type>|<storage type>

  131    4-18  <fixed type identifier> ::= <identifier>
                                          |<pre-defined type identifier>

  132     5-2  <floating point constant> ::= <real constant>
                              | <longreal constant>

  133    11-2  <floating point element> ::=
                 <floating point expression> [<floating point field specifier>]

  134    11-2  <floating point expression> ::= <real expression>
                                         | <longreal expression>

  135    11-2  <floating point field specifier> ::=
                         : <field length> [:<fractional digits>]

  136     4-7  <floating point type> ::= <real type> | <longreal type>

  137     3-3  <follower> ::= <letter>|<digit>
                              |_|#|$|@

  138    10-6  <for list> ::=
                     <initial value> to <final value>
                    |<initial value> downto <final value>

  139    10-6  <for statement> ::=
                         for <control variable> := <for list> do
                             <statement list> forend

  140     8-2  <formal param list> ::= <formal parameter identifier>
                                {,<formal parameter identifier>}

  141     8-2  <formal parameter identifier> ::= <identifier>
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


 142     11-2  <fractional digits> ::= <positive integer expression>

 143    10-21  <free statement> ::=
                    free <pointer variable>[in <heap variable>]

 144      8-3  <func attribute> ::= <proc attribute> | unsafe

 145      8-3  <func body> ::= <proc body>

 146      8-3  <func end> ::= funcend [<function identifier>]

 147      8-2  <func spec> ::= <function identifier> [<alias>] <func type spec>

 148      8-3  <func type spec> ::= [<parameter list>] : <result type>

 149      8-2  <function declaration> ::= function [ xref ] <func spec>
                         | function [[ func attribute]] <func spec> ;
                                   <func body> <func end>

 150      8-3  <function identifier> ::= <identifier>

 151      9-1  <function reference> ::= <built-in function reference>
                                        |<user defined function reference>

 152     4-22  <function type> ::= <function type identifier>

 153     4-23  <function type identifier> ::= <identifier>

 154      6-6  <global proc name> ::= <procedure identifier>

 155     4-18  <heap type> ::= heap (<space>)
                             | <heap type identifier>

 156     4-18  <heap type identifier> ::= <identifier>

 157    10-20  <heap variable> ::= <variable reference>

 158      5-2  <hex digit> ::= A|B|C|D|E|F
                              |a|b|c|d|e|f
                              |<digit>

 159      3-3  <identifier> ::= <letter>{<follower>}

 160     10-9  <if body> ::= <expression> then <statement list>
                   [ else <statement list> | elseif <if body>]
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION

 161     10-9  <if statement> ::=
                  if <if body> ifend

 162      5-4  <indefinite value constructor> ::=
                                         [<value elements>]
                                       | [ ]   {the empty set}

 163     4-13  <index> ::= <scalar type>
                  |<constant scalar expression>
                          ..<constant scalar expression>

 164     10-6  <initial value> ::= <scalar expression>

 165      6-6  <initialization> ::= := <initialization expression>

 166      6-6  <initialization expression> ::= <constant expression>
                                  | <indefinite value constructor>
                                  | ↑<global proc name>

 167      5-2  <integer> ::= <digit>{<digit>}
                           | <digit>{<hex digit>}<base designator>

 168      5-1  <integer constant> ::= <integer> | <integer constant identifier>

 169      5-2  <integer constant identifier> ::= <identifier>

 170      4-3  <integer type> ::= integer|<integer type identifier>

 171      4-4  <integer type identifier> ::= <identifier>

 172     4-14  <invariant record spec> ::=
                         record <fixed fields> <recend>

 173     4-14  <invariant record type> ::=
                   [packed] <invariant record type identifier>
                  |[packed] <invariant record spec>

 174     4-14  <invariant record type identifier> ::= <identifier>

 175     12-6  <layout control> ::= <source layout>
                                  | <listing layout>

 176     12-6  <left> ::= <integer>

 177     4-12  <length> ::= <positive integer constant expression>
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


 178    10-16  <length fixer> ::= <non-negative integer expression>

 179      3-3  <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

 180     12-7  <lineation> ::= spacing  := <spacing>
                             | skip  := <number of lines>

 181     12-6  <listing layout> ::= <pagination>
                                  | <lineation>
                                  | <titling>

 182     12-5  <listing toggles> ::= list | listobj
                                   | listcts | listext | listall

 183      5-2  <longreal constant> ::= <longreal number>
                                     | <longreal constant identifier>

 184      5-2  <longreal constant identifier> ::= <identifier>

 185     11-2  <longreal expression> ::= <expression>

 186      5-2  <longreal number> ::= <mantissa> D<exponent>

 187      4-7  <longreal type> ::= longreal |<longreal type identifier>

 188      4-7  <longreal type identifier> ::= <identifier>

 189      4-6  <lower> ::= <constant scalar expression>

 190     4-20  <lower bound spec> ::= <constant integer expression>

 191     12-8  <maintenance control> ::= compile | nocompile

 192      5-2  <mantissa> ::= <digit>{<digit>}[.]{<digit>}

 193      7-1  <module body> ::= <declaration list>

 194      7-1  <module declaration> ::=
                    module <module identifier> [<alias>];
                      <module body>
                    modend [<module identifier>]
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

NUMBER	PAGE	CYBIL METALANGUAGE DEFINITION
195	7-1	<module identifier> ::= <identifier>
196	9-1	<multiplying operator> ::= * | div | / | mod | and
197	10-19	<next statement> ::= next <allocation designator> in <pointer to sequence referenc
198	6-11	<non-negative integer expression> ::= <scalar expression>
199	12-7	<number of lines> ::= <integer> {where 1 <= number of lines}
200	12-8	<object library control> ::= library := <library name>
201	4-9	<object type> ::= <type>
202	4-24	<offset> ::= <integer constant>
203	5-1	<ordinal constant> ::= <ordinal constant identifier>
204	4-4	<ordinal constant identifier list> ::= <ordinal constant identifier> ,<ordinal constant identifier> {,<ordinal constant identifier>}
205	4-4	<ordinal constant identifier> ::= <identifier>
206	4-4	<ordinal type> ::= (<ordinal constant identifier list>) | <ordinal type identifier>
207	4-4	<ordinal type identifier> ::= <identifier>
208	12-6	<pagination> ::= eject
209	8-2	<param> ::= <formal param list> : <parameter type>
210	8-2	<param segment> ::= <reference params> | <value params>
211	8-2	<parameter list> ::= (<param segment> {;<param segment>})
212	8-2	<parameter type> ::= <fixed type> |<adaptable type>

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


 213      9-2  <parental> ::= <parental type variable>


 214      4-9  <parental type> ::= <storage type>
                                  | <adaptable storage type>
                                  | <aggregate type>
                                  | <adaptable aggregate type>


 215      9-2  <parental type variable> ::= <variable>

 216      5-2  <pointer constant> ::= nil

 217     11-2  <pointer element> ::=
                         <pointer reference>[<pointer field specifier>]

 218     11-2  <pointer field specifier> ::= [:<field length>][:<radix spec>]

 219      6-9  <pointer reference> ::= <pointer variable>
                                      |<function reference>

 220      4-9  <pointer to cell> ::= ↑cell

 221      4-8  <pointer to function> ::= ↑<function type>


 222      4-8  <pointer to procedure> ::= ↑<procedure type>


 223     10-7  <pointer to procedure reference> ::= <pointer reference>

 224    10-19  <pointer to sequence reference> ::= <pointer to sequence variabl
                                                | <function reference>

 225    10-19  <pointer to sequence variable> ::= <pointer variable>

 226      4-8  <pointer type> ::= <fixed pointer>
                                 |<fixable pointer>
                                 |<pointer to procedure>
                                 |<pointer to function>
                                 |<pointer type identifier>

 227      4-8  <pointer type identifier> ::= <identifier>

 228      6-9  <pointer variable> ::= <variable>
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  229    4-18  <positive integer constant expression> ::=
                    <constant scalar expression>

  230    12-3  <pragmat> ::= <toggle control>
                           | <layout control>
                           | <maintenance control>
                           | <comment control>
                           | <object library control>

  231    12-3  <pragmat statement> ::=
                         ?? <pragmat> { ,<pragmat> } ??

  232    4-18  <pre-defined type identifier> ::= integer | boolean | char
                                               | real | longreal | cell

  233     8-2  <proc attribute> ::= xdcl | inline | #gate

  234     8-2  <proc attributes> ::= <proc attribute> , {<proc attribute>}

  235     8-2  <proc body> ::= <declaration list> <statement list>

  236     8-2  <proc end> ::= procend [<procedure identifier>]

  237     8-2  <proc spec> ::= <procedure identifier> [<alias>] <proc type spec>

  238     8-2  <proc type spec> ::= [<parameter list>]

  239    10-7  <procedure call statement> ::=
                    <procedure reference> <actual parameter list>

  240     8-1  <procedure declaration> ::=
                      procedure [ xref ] <proc spec>
                    | procedure[[<proc attributes>]]<proc spec>;
                              <proc body><proc end>
                    | program <proc spec>;<proc body><proc end>

  241     8-2  <procedure identifier> ::= <identifier>

  242    10-7  <procedure reference> ::= <procedure identifier>
                                       | <pointer to procedure reference> ↑

  243    4-22  <procedure type> ::= <procedure type identifier>
                                  |procedure <proc type spec>

  244    4-22  <procedure type identifier> ::= <identifier>
```

NUMBER	PAGE	CYBIL METALANGUAGE DEFINITION
245	10-18	<push statement> ::= push <allocation designator>
246	5-2	<radix> ::= 2 | 8 | 10 | 16
247	11-2	<radix spec> ::= #(<radix>)
248	5-2	<real constant> ::= <real number> | <real constant identifier>
249	5-2	<real constant identifier> ::= <identifier>
250	11-2	<real expression> ::= <expression>
251	4-7	<real type> ::= real |<real type identifier>
252	4-7	<real type identifier> ::= <identifier>
253	4-14	<recend> ::= [,]recend
254	6-15	<record subreference> ::= <field selector>|<subscripted reference>
255	4-14	<record type> ::= <invariant record type> |<variant record type>
256	8-2	<reference params> ::= var <param> { ,<param> }
257	9-1	<relational operator> ::= < | <= | > | >= | = | <> | in
258	4-9	<relative pointer type> ::= rel (<parental type>) ↑ <object type>
259	5-4	<rep spec> ::= rep <positive integer constant expression> of
260	10-5	<repeat statement> ::= repeat <statement list> until <expression>
261	10-19	<reset statement> ::= reset <pointer to sequence variable> [to <pointer reference>] | reset <heap variable>
262	8-3	<result type> ::= <basic type>

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  263   10-14  <return statement> ::= return

  264    12-6  <right> ::= <integer>

  265     5-1  <scalar constant> ::= <ordinal constant>
                                   | <boolean constant>
                                   | <integer constant>
                                   | <character constant>

  266    11-2  <scalar element> ::=
                    <scalar expression>[<scalar field specifier>]

  267    10-6  <scalar expression> ::= <expression>

  268    11-2  <scalar field specifier> ::=
                          [:<field length>] [:<radix spec>]

  269    4-11  <scalar identifier> ::= <identifier>

  270     4-3  <scalar type> ::= <integer type>
                               |<character type>
                               |<ordinal type>
                               |<boolean type>
                               |<subrange type>

  271     9-2  <scalar type identifier> ::= <scalar identifier>

  272     9-2  <scalar variable> ::= <variable>


  273     5-2  <scaled number> ::= <mantissa> E<exponent>

  274     6-5  <scope attribute> ::= xdcl | xref | #gate

  275     6-7  <section attribute> ::= read | write

  276     6-7  <section declaration> ::= section <sections> {,<sections>}

  277     6-7  <section name> ::= <identifier>

  278     6-7  <sections> ::=
                  <section name> {,<section name>} : <section attribute>

  279    4-16  <selection spec> ::= <constant scalar expression>
                                  [..<constant scalar expression>]
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


 280    10-10  <selection spec> ::=
                            <constant scalar expression>
                          [..<constant scalar expression>]

 281     4-16  <selection specs> ::= <selection spec>
                                     {, <selection spec>}

 282    10-10  <selector> ::= <scalar expression>

 283     4-17  <sequence type> ::= seq (<space>)
                                 | <sequence type identifier>

 284     4-18  <sequence type identifier> ::= <identifier>

 285     4-11  <set type> ::= set of <base type>
                            |<set type identifier>

 286     4-11  <set type identifier> ::= <scalar identifier>

 287      5-4  <set value constructor> ::=
                    $<set type identifier> [ ]    {the empty set}
                  | $<set type identifier> [ <set value elements>]

 288      5-4  <set value element> ::= <expression>

 289      5-4  <set value elements> ::= <set value element>
                                        {,<set value element>}

 290      9-1  <sign> ::= + | -

 291      9-1  <sign operator> ::= <sign>

 292      9-1  <simple expression> ::= <term> | <sign operator><term>
                                     |<simple expression>
                                        <adding operator><term>

 293     12-6  <source layout> ::= <source margin control>

 294     12-6  <source margin control> ::=  left  := <left>
                                         |  right  := <right>

 295     4-18  <space> ::= <fixed span>{,<fixed span>}

 296     12-7  <spacing> ::= 1 | 2 | 3
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  297    10-16 <span> ::= [rep <non-negative integer expression> of]
                    <fixed type identifier>

  298    10-16 <span fixer> ::= [<span> {, <span> }]

  299      3-3 <special mark> ::= +|-|*|/|.|;|:|,
                                 |#|$|_|@|?|(|)|=|<|>|[|]|↑|{|}

  300    10-16 <star fixer> ::= <scalar expression> ..  <scalar expression>

  301    10-1  <statement> ::= <assignment statement>
                              |<structured statement>
                              |<control statement>
                              |<storage management statement>

  302    10-1  <statement list> ::= <statement>{;<statement>}

  303      6-4 <storage attribute> ::= static | <section name>

  304    10-15 <storage management statement> ::= <push statement>
                                                 |<next statement>
                                                 |<reset statement>
                                                 |<allocate statement>
                                                 |<free statement>

  305     4-17 <storage type> ::= <sequence type>
                                 |<heap type>

  306      5-2 <string constant> ::= <string term>
                                      { cat <string term>}

  307      9-2 <string constant identifier> ::= <identifier>


  308      3-3 <string delimiter> ::= '

  309     11-2 <string element> ::=
                    <string expression> [<string field specifier>]

  310     11-2 <string expression> ::= <string variable>
                                     | <string constant>
                                     | <substring reference>

  311     11-2 <string field specifier> ::= :<field length>
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


 312      5-2  <string term> ::= <character constant>
                         |'[<char token> <char token> {<char token>}]'

 313     4-12  <string type> ::= <fixed string>
                                  | <string type identifier>

 314     4-12  <string type identifier> ::= <identifier>

 315     6-10  <string variable> ::= <variable reference>

 316     10-4  <structured statement> ::= [<structured statement designator>]
                                           <repeat statement>
                         |[<structured statement designator>] <delimited statement>
                                           [<structured statement designator>]

 317     10-4  <structured statement designator> ::=
                              / <structured statement identifier> /

 318     10-4  <structured statement identifier> ::= <identifier>

 319     4-10  <structured type> ::= <set type>
                                    |<aggregate type>

 320      4-6  <subrange type> ::= <subrange type identifier>
                                  |<lower>..<upper>

 321      4-6  <subrange type identifier> ::= <identifier>

 322     6-13  <subscript> ::= <scalar expression>

 323     6-13  <subscripted reference> ::= <array variable> [<subscript>]

 324     6-11  <substring length> ::= <non-negative integer expression>
                                     | *


 325     6-10  <substring reference> ::=
                         <string variable>(<substring spec>)

 326     6-10  <substring spec> ::=
                         <first char>[,<substring length>]

 327    10-16  <tag field fixers> ::= <scalar expression>
                         | <constant fixers>[,<scalar expression>]
```

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION

 328     4-15  <tag field selector> ::= <identifier>

 329     4-15  <tag field spec> ::= [<tag field selector> : ] <tag field type>

 330     4-15  <tag field type> ::= <scalar type>

 331      9-1  <term> ::= <factor>
                       | <term><multiplying operator><factor>

 332     12-1  <text> ::= <text item> {<text item>}

 333     12-1  <text item> ::= <pragmat statement>
                            | <compile-time statement>
                            | <identifier>
                            | <constant>
                            | <basic symbol other than ??>
                            | <comment>

 334     12-7  <titling> ::=
                      newtitle  := '<char token> {<char token>}'
                     | title  := '<char token> {<char token>}'
                     | oldtitle

 335     12-4  <toggle control> ::= set (<toggle setting list>)
                                  | push (<toggle setting list>)
                                  | pop
                                  | reset

 336     12-4  <toggle identifiers> ::= <listing toggles>
                                      | <checking toggles>

 337     12-4  <toggle setting> ::= <toggle identifiers> := <condition>
                                | <empty>

 338     12-4  <toggle setting list> ::= <toggle setting> {,<toggle setting>}

 339      4-1  <type> ::= <fixed type>
                        |<fixable type>
                        |<procedure type>

 340      4-2  <type declaration> ::=
                         type <type spec>{, <type spec>}

 341      4-2  <type spec> ::= <identifier> = <type>
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


 342      5-2  <unscaled number> ::= <digit> {<digit>}.  <digit>{<digit>}

 343      3-3  <unused mark> ::= &|%|⊥|¬|¢|\|!|"

 344      4-6  <upper> ::= <constant scalar expression>

 345      9-1  <user defined function reference> ::=
                         <function identifier>(<actual parameter>
                         {, <actual parameter>})
                         | <function identifier>()

 346      5-4  <value element> ::=
                        [<rep spec>]<initialization expression>
                      | [<rep spec>]<set value constructor>
                      | [<rep spec>]<indefinite value constructor>
                      | [<rep spec>] *

 347      5-4  <value elements> ::=
                         <value element>{,<value element>}

 348      8-2  <value param> ::= <formal param list> : <parameter type>

 349      8-2  <value params> ::= <value param>{,<value param>}

 350      6-8  <variable> ::= <variable reference>
                             |<substring reference>

 351      6-3  <variable declaration> ::=
                   var <variable spec>
                     {,<variable spec>}

 352      6-3  <variable identifier> ::= <identifier>

 353      6-3  <variable identifiers> ::=
                     <variable identifier> [<alias>]
                     {,<variable identifier>[<alias>]}

 354      6-8  <variable reference> ::= <variable identifier>
                                       |<pointer reference>↑
                                       |<subscripted reference>
                                       |<field reference>

 355      6-3  <variable spec> ::=
                     <variable identifiers> : [<attributes>]
                     <fixed type>[<initialization>]
```

APPENDIX A - CYBIL METALANGUAGE CROSS-REFERENCE

```
NUMBER   PAGE  CYBIL METALANGUAGE DEFINITION


  356    4-16  <variant> ::= [<fixed fields>]
                            |[<fixed fields>,] <case part>

  357    4-15  <variant record spec> ::=
                    record [<fixed fields>,] <case part> <recend>

  358    4-15  <variant record type> ::=
                      [<packed>] <variant record type identifier>
                     |[<packed>] <variant record spec>

  359    4-15  <variant record type identifier> ::= <identifier>

  360    4-15  <variation> ::= =<selection specs>= <variant>

  361    4-15  <variations> ::= <variation> {, <variation>}

  362    10-5  <while statement> ::=
                    while <expression> do <statement list> whilend
```

APPENDIX B - CYBIL RESERVED WORD LIST

LINE	A1 LINE	X-REF	RESERVED WORD	LINE	A1 LINE	X-REF	RESERVED WORD
1	274	6-5	#gate	45	139	10-6	do
2	233	8-2	#gate	46	138	10-6	downto
3	61	9-2	#loc	47	208	12-6	eject
4	61	9-2	#ptr	48	160	10-9	else
5	61	9-2	#rel	49	83	12-2	else
6	61	9-2	#seq	50	160	10-9	elseif
7	61	9-2	#size	51	51	10-5	end
8	69	5-1	$char	52	110	10-13	exit
9	61	9-2	$char	53	53	5-1	false
10	61	9-2	$integer	54	82	12-2	false
11	61	9-2	$longreal	55	139	10-6	for
12	61	9-2	$real	56	139	10-6	forend
13	33	7-4	alias	57	143	10-21	free
14	34	4-24	aligned	58	146	8-3	funcend
15	35	10-20	allocate	59	149	8-2	function
16	196	9-1	and	60	155	4-18	heap
17	85	12-2	and	61	14	4-22	heap
18	38	4-13	array	62	161	10-9	if
19	10	4-20	array	63	83	12-2	if
20	51	10-5	begin	64	161	10-9	ifend
21	55	4-5	boolean	65	257	9-1	in
22	232	4-18	boolean	66	197	10-19	in
23	86	12-1	boolean	67	35	10-20	in
24	60	4-23	bound	68	143	10-21	in
25	62	4-15	case	69	233	8-2	inline
26	63	10-9	case	~~70~~	~~170~~	~~4-3~~	~~integer~~
27	64	4-16	casend	71	232	4-18	integer
28	306	5-2	cat	72	294	12-6	left
29	66	4-7	cell	73	200	12-8	library
30	220	4-9	cell	74	182	12-5	list
31	232	4-18	cell	75	182	12-5	listall
32	71	4-4	char	76	182	12-5	listcts
33	232	4-18	char	77	182	12-5	listext
34	73	12-5	chkall	78	182	12-5	listobj
35	73	12-5	chknil	79	187	4-7	longreal
36	73	12-5	chkrng	80	232	4-18	longreal
37	73	12-5	chksub	81	61	9-2	lowerbound
38	73	12-5	chktag	82	61	9-2	lowervalue
39	75	12-8	comment	83	196	9-1	mod
40	191	12-8	compile	84	194	7-1	modend
41	92	5-3	const	85	194	7-1	module
42	101	10-12	cycle	86	334	12-7	newtitle
43	196	9-1	div	87	197	10-19	next
44	362	10-5	do	88	216	5-2	nil

APPENDIX B - CYBIL RESERVED WORD LIST

LINE	A1 LINE	X-REF	RESERVED WORD	LINE	A1 LINE	X-REF	RESERVED WORD
89	191	12-8	nocompile	133	276	6-7	section
90	113	9-1	not	134	283	4-17	seq
91	82	12-2	not	135	21	4-21	seq
92	285	4-11	of	136	285	4-11	set
93	62	4-15	of	137	335	12-4	set
94	127	4-18	of	138	180	12-7	skip
95	259	5-4	of	139	180	12-7	spacing
96	63	10-9	of	140	303	6-4	static
97	297	10-16	of	141	128	4-12	string
98	90	12-4	off	142	13	4-20	string
99	334	12-7	oldtitle	143	61	9-2	strlength
100	90	12-4	on	144	61	9-2	succ
101	31	9-1	or	145	160	10-9	then
102	106	12-2	or	146	83	12-2	then
103	39	4-13	packed	147	334	12-7	title
104	173	4-14	packed	148	138	10-6	to
105	358	4-15	packed	149	261	10-19	to
106	6	4-20	packed	150	53	5-1	true
107	17	4-21	packed	151	82	12-2	true
108	60	4-23	packed	152	340	4-2	type
109	335	12-4	pop	153	144	8-3	unsafe
110	61	9-2	pred	154	260	10-5	until
111	243	4-22	procedure	155	61	9-2	upperbound
112	240	8-1	procedure	156	61	9-2	uppervalue
113	236	8-2	procend	157	351	6-3	var
114	240	8-1	program	158	256	8-2	var
115	245	10-18	push	159	80	12-1	var
116	335	12-4	push	160	362	10-5	while
117	275	6-7	read	161	362	10-5	whilend
118	251	4-7	real	162	275	6-7	write
119	232	4-18	real	163	274	6-5	xdcl
120	253	4-14	recend	164	233	8-2	xdcl
121	172	4-14	record	165	31	9-1	xor
122	357	4-15	record	166	106	12-2	xor
123	19	4-21	record	167	274	6-5	xref
124	258	4-9	rel				
125	127	4-18	rep				
126	259	5-4	rep				
127	297	10-16	rep				
128	260	10-5	repeat				
129	261	10-19	reset				
130	335	12-4	reset				
131	263	10-14	return				
132	294	12-6	right				

APPENDIX C - CYBIL INTRINSICS

GENERAL INTRINSICS

The following intrinsics are considered useful across a wide variety of processors where CYBIL is provided.

#CONVERT_POINTER_TO_PROCEDURE (P,Q)

This procedure is used to convert a variable of type pointer-to-procedure with no parameters to a variable of type pointer-to-procedure with an arbitrary parameter list.

P - pointer-to-procedure with no parameters

Q - pointer-to-procedure with an arbitrary parameter list.

#KEYPOINT (P1,P2,P3)

This procedure causes a KEYPOINT instruction (Reference Number 136) to be generated based on the following parameters:

P1 - This parameter specifies the keypoint class and is a constant expression in the range 0..15 and becomes the instruction J field.
P2 - This parameter specifies optional data to be collected with the keypoint and is a constant or variable expression within the range 0..0fffffff(16). If it is the constant zero then the K field of the instruction is zero. If P2 is not a zero then the value of the P2 is placed in an X register and that register number becomes the instruction's K field.
P3 - This parameter specifies a keypoint identifier and is a constant expression in the range of 0..0FFFF(16) and becomes the instructions Q field.

#SCAN (SELECT, STRING, INDEX, FOUND)

This procedure scans a string from left to right until either one of a set of specified characters is found or until the string is exhausted. The set of character values to scan for is specified with a 256 bit variable, with each bit representing one of the possible character values. If a bit is set in this variable, the scan will stop when a character value corresponding to the bit position in the variable is found. In either termination case, the starting character position of the character that caused termination is returned. The procedure returns a boolean which indicates if a byte was "found".

select - Variable designating the character values to be scanned for. The size of this variable must be 256 bits.
string - String or substring variable to be scanned
index - Integer variable (1..65536) into which the index of a "found" character is returned. If no selected values were found, it contains the string length plus one. (The index value of the first character in the string is one.)
found - Boolean variable which is set to true if the scan terminated as a result of finding one of the selected characters.

#SPOIL (,VARIABLE>{,<VARIABLE>})

This procedure is used to announce to the compiler that certain optimizations should be inhibited on the quoted (up to a limit of 127) variables. This inhibited optimization is necessary to control asynchronous usage of CYBIL. The compiler will handle each actual parameter to #SPOIL as if it was associated with a reference (VAR) formal parameter.

If the parameter quoted is a direct reference to a variable, it will be assumed to interfere with that variable. If the parameter quoted is an indirect reference (i.e. pointer dereference, records with pointer fields) to a variable, it will be assumed to interfere with any variable of equivalent type.

#TRANSLATE (TABLE, SOURCE, DESTINATION)

This procedure translates each character contained in the source field, according to the translation table, and transfers the results to the destination field. The translation operation will occur from left to right with each source byte used as an index into the translation table. Translated bytes obtained from the translation table are stored into the destination field. If the length of the source field is less than the length of the destination field, translated blanks will be used to fill the destination field. If the length of the source field is greater than the length of the destination field, rightmost characters of the source field will be truncated.

table - string variable with length 256 that defines the translation table.
source - string expression to be translated
destination - string variable or substring reference into which the

translated string is transferred.

#UNCHECKED_CONVERSION (SOURCE, TARGET) used to assign values of different type

This procedure copies SOURCE to TARGET. The following restrictions must be satisfied:

1) SOURCE and TARGET must be <variable reference>s

2) SOURCE and TARGET must be of the same length as measured in bits

3) if SOURCE or TARGET is a <pointer reference>↑ then the <pointer reference> must not be a <pointer to procedure>

4) TARGET must satisfy the restrictions on the target of an assignment statement

5) neither SOURCE not TARGET can be a pointer or contain a pointer.

MACHINE SPECIFIC INTRINSICS

C180 INTRINSICS

The following intrinsics are provided for the CYBIL implementation on the Advanced System. These intrinsics allow system programmers access, in CYBIL, to a small subset of the hardware instructions and data structures.

key-expression new-lock-expression old-lock-variable

#COMPARE SWAP (LOCK, INITIAL, NEW, ACTUAL, RESULT)

This procedure externalizes the compare swap (Reference Number 125) instruction. The operation of this procedure can best be described with the CYBIL statements given below. Note that the hardware executes the entire statement list as a non-interruptable sequence and that access to LOCK from other sources (other processor, PPU) is prevented during the time it takes to execute the statement list.

```
If (left half of lock) = OFFFFFFFF(16) THEN
  result := 2;
ELSE
  actual := lock;
  If lock = initial THEN
    lock := new;
    result := 0
    ELSE
      result := 1;
    IFEND
  IFEND
```

lock - Variable on which the compare swap operation is to be performed. This variable must be on a [0 mod 8] boundary.
initial - Expression that specifies what the initial content of lock must be for the swap operation to be successful.
new - Expression that specifies the value to be stored in lock if the swap is successful.
actual - Variable into which the initial contents of lock is returned. If lock is locked, then actual is not modified.
result - Variable 0..2 into which the result of the compare_swap instruction is returned.
 0 - swap was successful.
 1 - swap failed because initial <> actual
 2 - swap failed because variable was locked.

The TYPE of lock, initial, new, and actual must be equivalent and have a size of 8 bytes.

#CALLER ID (ID)

i.e. proc; [illegible] caller id ( )

This procedure obtains the id of the caller of the function or procedure. Caller ID is placed in X0 left by the hardware as a result of executing a CALLREL or CALLSEG instruction. The caller id is a record that contains the global/local key, ring, and segment number of the caller of a procedure. The argument to this procedure can be any record with a size of 4 bytes. See sections 2.1.1.1, 2.6.1.2 and 2.6.1.3 of the CYBER 180 MIGDS for a complete description of the caller id.

#HASH SVA (SVA, INDEX, COUNT, FOUND)

This procedure externalizes the LPAGE (Reference Number 127) instruction. This instruction searches the System Page Table (SPT) for a specified System Virtual Address (SVA) and returns an index to the entry (if found) or an index to the last entry searched (if not found). A count of the number of entries searched is also returned. This procedure returns a boolean to indicate whether the SVA was found.

sva - variable that contains the SVA to search for. The size of this variable must be 6 bytes.
index - Integer variable to which a word index into the System Page

Table is returned. This index points to the SPT entry for the SVA if the SVA was found, or to the last entry searched if the entry was not found.
count - integer variable (1..32) to which a count is returned of the number of SPT entries searched.
found - boolean variable that specifies whether the SVA was found.

#RING (POINTER): INTEGER

This function takes a direct pointer expression and returns an integer value which is the ring number contained in the pointer.

#SEGMENT (POINTER): INTEGER

This function takes a direct pointer expression and returns an integer value which is the segment number contained in the pointer.

#OFFSET (POINTER): INTEGER

This function takes a direct pointer expression and returns an integer value which is the signed offset contained in the pointer.

#ADDRESS (RING, SEGMENT, OFFSET): ↑CELL

This function takes a ring, segment and offset and returns a value of type pointer to cell. The values for the arguments must be in the following ranges:

```
ring: 1..15
segment: 0..4095
offset: -80000000(16)..7fffffff(16)
```

#CURRENT STACK FRAME: ↑CELL

This function returns a pointer to the first cell of the current stack frame.

#PREVIOUS SAVE AREA: ↑CELL

This function returns a pointer to the first cell of the previous save area.

#PURGE BUFFER (OPTION, ADDRESS)

This procedure externalizes the PURGE (Reference Number 138) instruction for purging the contents of the cache or map.

option - constant integer expression in the range of 0 to 15 that specifies the purge option. See the MIGDS for a description of the values of the purge option.
address - a 6 byte variable that specifies the PVA or SVA of the data to be purged.

#TEST SET (VARIABLE, RESULT)

This procedure externalizes the LBSET (Reference Number 124) instruction to return a single bit from memory and to unconditionally set that bit in memory without changing the value of any other memory. This intrinsic works on a boolean variable reference whether it be a boolean variable, an array of booleans, a field of either a packed or unpacked record.

variable - This variable reference is for the boolean variable that the LBSET instruction operates on.
result - This variable reference is where the boolean result will be returned from the LBSET operation.

C180 AND C200 INTRINSICS

#FREE RUNNING CLOCK (CLOCK ID): INTEGER

This <u>unsafe</u> function returns the value of the free running microsecond clock.

clock_id - Integer expression (0 ..1) designating the clock to be read. (For the C180, this is the memory port to be used. For the C200, this value must be zero.)

#READ REGISTER (REGID): INTEGER

This <u>unsafe</u> function externalizes the reading of the specified register. This allows a program to read the contents of a process or processor register file. The result of the function is an integer.

regid - Integer expression (0 .. 255) that identifies the number of the register to be read.

#WRITE REGISTER (REGID, VALUE)

This procedure externalizes the changing of the content of the specified process or processor register file.

regid - Integer expression (0 .. 255) that identifies the number of the register to be written.
value - Integer expression that contains the data to be written to the register.

C200 INTRINSICS

#GET JOB TIMER : INTEGER

This unsafe function externalizes the RJTIME instruction (opcode=37) which retrieves the contents of the job interval timer. This intrinsic produces undefined results when issued in monitor mode.

#LOAD AR

This procedure externalizes the LODAR instruction (opcode=0D) which loads the associative registers from absolute bit address 4000(16) in conjunction with #SPOIL as appropriate.

#SET JOB TIMER (TIME)

This procedure externalizes the WJTIME instruction (opcode=3A) which sets a value into the job interval timer. When executed in Monitor Mode this intrinsic is a no-op.
time - Integer expression whose contents is set into the job interval timer. If the value is greater that (2**32)-1, the high order bits will be truncated. If time =0 then the job interval timer is de-activated.

#STORE AR

This procedure externalizes the STOAR instruction (opcode=0C) which stores the associative registers into central processor memory at absolute bit address 4000(16). The contents of the live associative registers are undefined after completion of the store. This procedure will be used in conjunction with #SPOIL as appropriate.

#SWAP DFBR (CURRENT REGISTER, NEW REGISTER)

This procedure externalizes the LSDFR instruction (opcode=3B) which loads a new value (new_register) into the 64-bit "data flag branch register" while storing the old contents of this register into the variable (current_register). Note: An immediate data flag branch will occur at the completion of this intrinsic if the new contents of the DFBR meet the appropriate branch conditions.

current_register - 64-bit, word aligned variable which will receive the old contents of the "data flag branch register".

new_register - 64-bit, word aligned integer expression or variable which contains the new value to be loaded into the live "data flag branch register".