2.0 LANGUAGE OVERVIEW

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

The definable elements include variables, procedures, labels and files,* all having the characteristics that are conventionally associated with their names. Declarations of instances of these elements 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 referential notation. 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., an integer may not be added to a real number). 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, boolean, and real types, all having their conventional connotations, value sets, and operational domains. The first three are scalar types, which define well-ordered sets of values -- as distinguished from real types. 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. Pointer types are

* Release 1 does not support files.

included in the basic types. They represent location values, and other descriptive information, that can be used to reference instances of variables and other SWL elements. Pointers are always bound to a specific type, and pointer variables may assume, as values, only pointers to elements of that type.

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. Five structuring methods are available: set structure, string structure, array structure, record structure and union structures.

A set type represents the subset 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 ($1 \leqslant K \leqslant n$) is called a sub-string. Notation for accessing sub-strings 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. 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.

~~A union type represents a finite set of selectable, non-equivalent types. Union types permit one to define procedures whose parameters can be of more than one type and provide an alternative to variant record types.~~

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. Crammed structured types are used to spell out the precise representation of a structure in terms of the bit-lengths and relative alignments of its components. The use of crammed types is restricted to the so-called representation-dependent portion of a program.~~

Storage types represent structures to which other variables may be added, referenced and deleted under explicit program control. ~~They are also the only types that can be used to define relative pointers, which can be used to reference variables added to storage structures.~~ There are three storage types, each with its own management and access characteristics. A stack type represents a collection of components of the same type which is accessed by a "last in-first out" discipline. The "top" component of a stack can be referenced by using the stack's identifier as a pointer. Sequence types and heap types represent storage structures whose components may be of diverse type. Components of sequences must be accessed by a sequential accessing discipline (thru the operations of "resetting" to the first component and moving to the next component). Space for components of heap storages must be explicitly managed by the operation of allocate and free; the components are accessed thru pointers constructed as by-products of the allocate operation. The only storage type supported by Release 1 is the default heap.

Many of the structured and storage types (and subrange types) are described in terms of attributes, called bounds, that specify their shapes and extents. If the values of such attributes can be determined by a perusal of the entire program, then the associated type is precisely defined, and is said to be of fixed type; otherwise, the type is said to be of variable bound type. In the latter case, the type represents a class of potential instances of fixed types. An "instantaneous" fixed type for these is established whenever the type declaration is elaborated during execution (upon entering the block in which the declaration occurs), and persists over the scope of the declaration. Variable bound types are not supported by Release 1.

Adaptable types are array, string, record and storage types defined in terms of one or more indefinite bounds. 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 allocated – in which case the actual bounds are specified in the allocate statement. Adaptable types are not supported by Release 1.

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

Definite value constructors, which include specific type information, may be used freely in expressions. Indefinite value constructors can be used only 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 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 grouped together as either begin-end blocks or procedures. The former are used primarily to define scope and provide shielding. The latter also have identifiers associated with them, so that the identified portions of the program can be activated on demand by statements of the language.

Procedures are declared in terms of their identifier, the associated program, a set of attributes, and a list of formal parameters. Formal parameters are variable declarations which 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. Two methods of parameter binding are provided - call-by-value and call-by-reference; they have their conventional connotations.

A function is a procedure that returns a value of a specified type. These return-types are restricted to the basic types, and are specified in the procedure declaration.

~~Procedures may be used in the creation of coprocesses, which are distinct synchronous processes. Instead of the entire procedure being executed and then returning in line, coprocesses allow partial execution of a set of procedures with the single thread of control being passed back and forth amongst them through the resume statement. Subsequent resumption of a coprocess causes execution to commence with the successor of the last executed resume statement of the coprocess.~~

~~Variables and procedures sharing common attributes can be associated with segments which are identified areas for the storage and management of the elements associated with the segment. Segments are defined by segment declarations, and segment associations are specified in variable and procedure declarations (as a specific attribute).~~

~~In addition to their other programmatic aspects, blocks (together with segments and attributes) 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 declared in terms of a grouped set of declarations and a list of identifiers for elements declared within the module that can be referenced from without the module. All other identifiers are blocked-off. Modules are primarily designed to permit program repackagings at the "source" language level.~~

Statements define actions to be performed. Structured statements are constructs composed of statement lists: begin statements provide for scope control and storage allocation for their constituent declarations;⋈ if statements provide for the conditional execution of one of a set of statement lists; loop statements cause unbounded repetitions of their statement list; while, for and repeat statements control repetitive execution of their statement lists; case statements conditionally select one of their component statement lists for execution; variant case statements allow access to the variant fields of records; ~~conformity case statements select one of their component statement lists for execution, depending on the type of the value of a union variable.~~

Control statements cause the creation or destruction of execution environments. They provide for the activation of procedures; the creation, resumption and destruction of coprocesses; and general changes in the flow of control.

⋈ Begin statements are not supported by Release 1.

Storage management statements provide mechanisms for pushing and popping stack components, moving forward and backward over components of sequences, and allocating and freeing storage for components of heaps.

Finally, assignment statements cause variables to assume new values.

A SWL program is meant to be translated, by a compilation process into a SWL object program. Object programs resulting from distinct compilation can be combined by a linking process, into a single object program, and may undergo further transformation, by a loading process, into forms capable of direct interpretation (execution) by members of the IPL line.

~~Compile-time facilities, that are essentially extra-linguistic in nature, are used to control the compilation process and construct the program to be compiled. The facilities divide into two categories. The first category consists of the compile-time variable declarations, compile-time assignment statements, compile-time if statements, and macro facilities. The second category consists of the micro mechanism, which is applied to the text produced by the first category. The micro mechanism is analagous to the macro mechanism, but must follow normal block-structure rules for scope, and cannot affect the existing declaration or block structure.~~

~~Mechanisms for the incorporation of some representation-dependent facilities are provided. Their use may be dependent on the SWL compiler's allocation algorithms and on the target hardware design. The use of these facilities is restricted to procedures declared with the repdep attribute. The facilities include a cell type, that represents the smallest unit of directly addressable storage, crammed types which are memory-dependent structures with specified component bit-sizes and alignments, and methods for overriding pointer-to-type equivalence restrictions.~~

~~An extended set of machine dependent facilities including native data types, storage attributes and instructions, are to be provided for each machine for which SWL will generate object codes. The use of such facilities is restricted to the body of the so-called code statement which may include SWL statements and declarations as well as native instructions.~~

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. 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} .

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

3.2 BASIC 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>
        <alphabet> ::= <letter>
                     | <digit>
                     | <special mark>
                     | <unused mark>

<letter> ::= A|B|C|D|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
<special mark> ::=  + | - | * | / | . | , | ; | : | " | '
                  | # | $ | _ | @| ? | ( | ) | = | < | >
                  | \ | [ | ] | ∧ | ! | b̸ |
<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 30 characters, and identifiers that differ only by case shifts of component letters are considered to be identical.

3.2.3 BASIC SYMBOLS

Selected identifiers, special marks, digraphs of special marks, and other polygraphs are reserved for specific purposes in the language; e.g., as operators, separators, delimitors, groupers. 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

```
<constant> ::= <basic constant> | <string constant>
<basic constant> ::= <scalar constant>
                   ~~| <compile time variable>      "c.f., Section 12.1"~~
                   | <real constant>
                   | <pointer constant>
<scalar constant> ::= <ordinal constant>
                    | <boolean constant>
                    | <integer constant>
                    | <character constant>
```

```
<ordinal constant> ::= <ordinal constant identifier> . "c.f., 4.2.1.1.3"
<boolean constant> ::= false | true | <boolean constant identifier>
<integer constant> ::= <integer> | <integer constant identifier>
<character constant> ::= '<alphabet>' | <character constant identifier>
<real constant> ::= <real number> | <real constant identifier>
<string constant> ::=  <string term>
                     | <string term> {cat <string term>}
<string term> ::= <character constant> | <string constant identifier>
                | $char (<integer>)    "c.f., Standard Functions, 11.2"
                | '<alphabet> <alphabet> {<alphabet>}'
<pointer constant> ::= nil
<ordinal constant identifier> ::= <identifier>
<boolean constant identifier> ::= <identifier>
<integer constant identifier> ::= <identifier>
<character constant identifier> ::= <identifier>
<real constant identifier> ::= <identifier>
<string constant identifier> ::= <identifier>
<pointer constant identifier> ::= <identifier>
<real number> ::=  <unscaled number>
                 | <scaled number>
<unscaled number> ::= <digit> {<digit>} . <digit> {<digit>}
<scaled number> ::= <unscaled number> E [<sign>] <digit> {<digit>}
<integer> ::=  <digit> {<digit>}
             | <digit> {<hex digit>} <base designator>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<hex digit> ::=  A | B | C D | E | F
              | a | b | c | d | e | f
              | <digit>
<base designator> ::= (<radix>)
<radix> ::= 2 | 4 | 8 | 16
<sign> ::= + | -
```

3.2.5 CONVENTIONS

Identifiers, reserved words and constants must not abut and must not contain embedded blanks. 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 character> ::= <any ascii character other than double-quote
                         and semicolon>
```

4.0 SWL TYPES

SWL provides four classes of programmatic objects of discourse:

```
<SWL type> ::=  <data type>
              | <adaptable type>   ⋈
              | <formal type>      ⋈
              | <file type>        ⋈
```

Broadly speaking: data types are used to define sets of values that can be assumed by SWL variables; adaptable types define data types that have indefinite attributes, are meant to be explicitly "type fixed" during execution, and – together with formal types – may be used as formal parameters of procedures, and must otherwise be referenced through a pointer mechanism; formal types are associated with procedures, coprocesses, and labels (c.f., Section 8.0); file types are primarily used in input-output operations. ⋈

⋈ Release 1 does not support adaptable, formal, or file types.

4.1 TYPE DECLARATIONS

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

Type declarations provide the mechanism for introducing new types.

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

<type spec> ::= <type identifier ~~list~~> = <SWL type>

~~<type identifier list> ::= <identifier list>~~

~~<identifier list> ::= <identifier>{,<identifier>}~~

<type identifier> ::= <identifier>

4.2 DATA TYPES

```
<data type> ::= <type>
<type> ::= <basic type> | <structured type> | <storage type>
```

For brevity's sake, data types will be referred to in the sequel as types; for clarity's sake, references to other SWL type-varietals will be spelled out completely. Basic types define components that may take on simple values, while structured types and storage types define collections of components.

4.2.0 FIXED AND VARIABLE_BOUND TYPES

Many of the types (particularly the structured and storage types) are couched in terms of attributes that are called "lengths" or "sizes" or "bounds" or "index ranges", depending on the specific type and on the context in which it is being discussed. If the values of such attributes can be determined by a perusal of the entire program, then the associated type is precisely defined, and is said to be of fixed type; otherwise, the type is said to be of variable bound type. In the latter case, the type represents a class of potential instances of fixed types. An "instantaneous" fixed type for these is established whenever the type declaration is elaborated during execution (upon entering the block in which the declaration occurs), and persists over the scope of the declaration (c.f., Scope of Identifiers, 5.2). For purposes of exposition, the constructs

<variable_bound type>

and

<fixed type>

are introduced, the latter denoting all types but the former. Release 1 does not support variable bound types.

4.2.1 BASIC TYPES

```
<basic type> ::=  <scalar type>
                | <real type>
                | <pointer type>
```

4.2.1.1 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>
```

4.2.1.1.1 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 (c.f., 4.2.1.1.5) defined by

~~-n1 .. n2~~ $-(2^{48} - 1) \ldots (2^{48} - 1)$.

~~where n1 and n2 denote implementation-dependent integers.~~

4.2.1.1.2 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 (c.f., 4.2.1.1.5) defined by

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

where "$char" denotes the mapping function from integer type onto character type (c.f., Standard Functions, 11.2).

4.2.1.1.3 Ordinal Type

```
<ordinal type> ::=   (<ordinal list>)
                   | <ordinal type identifier>
<ordinal list> ::= <identifier list>
<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 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 list.* Ordinal type specifications are restricted to appear only in type declarations.

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 (c.f., Standard Functions, 11.2). 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.

* In ISWL, two separately defined ordinal types are never considered to be equivalent.

4.2.1.1.4 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 equivalent to the ordinal type specified by the ordinal list

(_false_, _true_)

4.2.1.1.5 Subrange Type

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

A subrange type represents a subrange of the values of another scalar type, defined by a lower bound and an upper bound. The lower bound must not be greater than the upper bound and both must be of equivalent scalar types. ~~Subrange types may be of variable bound type (c.f., 4.2.0).~~

Two subrange types are equivalent if they have identical upper and lower bounds;* and an improper subrange type (i.e., one that spans its 'parent' range) is equivalent to its 'parent' type.

* In ISWL, two types are equivalent only if they represent the same instance of a type definition.

Example:

```
type non_negative_integer = 0 .. n2,
     letter = 'A' .. 'Z',
     color = (red, orange, yellow, green, blue),
     hot_color = red .. yellow,
     hue = red .. blue,
     range = -10 .. 10
```

Note that the subrange type, "hue", is an improper subrange of, and therefore equivalent to, its parent ordinal type, "color".

4.2.2 REAL TYPE

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

The range and precision of real type is implementation-dependent. Conversion functions between real and integer type are provided (c.f., Standard Functions, 11.2).

4.2.3 POINTER TYPE

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

```
<pointer type> ::=  <direct pointer type>
                    ~~| <relative pointer type>~~
```

```
<direct pointer type> ::= ∧<type>
                        | <adaptable pointer>
                        | <formal pointer>
                        | ∧<file type>
```

~~<relative pointer type> ::= rel [(<storage type>)] ∧<type>~~

```
<adaptable pointer> ::= ∧<adaptable type>
<adaptable pointer to string> ::= <adaptable pointer>
<adaptable pointer to array> ::= <adapter pointer>
<adaptable pointer to stack> ::= <adaptable pointer>
<adaptable pointer to sequence> ::= <adaptable pointer>
<adaptable pointer to heap> ::= <adaptable pointer>
<formal pointer> ::= ∧<formal type>
<pointer to label> ::= <formal pointer>
<pointer to procedure> ::= <formal pointer>
```

~~<pointer to coproc> ::= <formal pointer>~~

Direct ~~(relative)~~ pointer types are equivalent if they are defined in terms of equivalent SWL types (types).*

Direct ~~(relative)~~ pointer types represent locations ~~(relative locations)~~ of instances of objects of SWL type (components of objects of storage type).

~~Built-in mapping functions between direct pointers and relative pointers are provided (c.f., Standard Functions, 11.2).~~

* In ISWL, two types are equivalent only if they represent the same instance of a type definition.

4.3 STRUCTURED TYPES

Structured types represent collections of components, and are defined by describing their component types and indicating a so-called structuring method. These differ in the accessing discipline and notation used to select individual components. ~~Five~~ Four structuring methods are available: set structure, string structure, array structure, and record structure. ~~and union structures.~~ Each will be described in the sequel. Structured types may be of variable bound type (c.f., 4.2.0).

```
<structured type>::=  <set type>
                    | <string type>
                    | <array type>
                    | <record type>
                    ~~| <union type>~~
```

4.3.1 SET TYPE

```
<set type> ::=  set of <base type>
             | <set type identifier>
<base type> ::= <scalar type>
<set type identifier> ::= <identifier>
```

A set type represents the set of 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). ~~Its value will be implementation dependent, but no less than 256 (to accommodate set of char).~~ ISWL limits the number of elements to 60.

Set types are equivalent if they have equivalent base types.[*]

Example: The set, access, declared by

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

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

$access [] "the empty set"

$access [no_read]

$access [no_write]

$access [no_execute]

$access [no_read, no_write]

$access [no_read, no_execute]

$access [no_write, no_execute]

$access [no_read, no_write, no_execute].

where the notation $access [...]" denotes a value constructor (c.f., Value constructors, Section 5.1) for the set type, access.

4.3.2 STRING TYPES

```
<string type> ::= string (<length>) of <character_type>
                | <string type identifier>
<length> ::= <positive integer constant ~~expression~~>
<string type identifier> ::= <identifier>
```

A string type of length n represents all ordered n-tuples of values of character type. An ordered k-tuple of these values ($1 \le k \le n$) is called a sub-string. Notation for accessing sub-strings is provided (c.f., Variables and Variable Declaration, 7.0).

[*] In ISWL, two types are equivalent only if they represent the same instance of a type definition.

Two string types are equivalent when they have the same length.* In the case of a variable length, the length is determined when the declaration is elaborated.

4.3.3 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 <u>n</u> index values whose types are indicated by the indices in the definition. Theoretically, the time needed to select a component is independent of the set of index values, so that an array structure is an example of a so-called random-access structure.

```
<array type> ::=  [<packing>] <array type identifier>
               |  [<packing>] <array spec>
<array type identifier> ::= <identifier>
<array spec> ::= array [<indices>] of <component type>
<indices> ::= <index> { , <index>}
<index> ::= <scalar type>
<component type> ::= <type>
<packing> ::= <packing attributes>
```

Packing attributes are used to specify component storage space - component access time trade-offs (c.f., Packing and Alignment, 4.8).

If <u>n</u> indices are specified, then the array type has dimension <u>n</u>. Two array types are equivalent if they have the same packing and dimensions, have equivalent component types, and corresponding indices are of equivalent types.* For variable index ranges, the index type is defined by the values of its constituent expressions determined when the declaration is elaborated.**

* In ISWL, two types are equivalent only if they represent the same instance of a type definition.

** Release 1 does not support variable bound arrays.

Example:

```
type hotness = array [color] of non_negative_integer,
     token_code = array [char] of token_class,
     token_class = (alpha, numeric, specials, others),
     array1 = array [1 .. 100, 100 .. 200] of 100 .. 300,
     i1 = 1 .. 100,
     i2 = 100 .. 200,
     s1 = 100 .. 300,
     array2 = array [i1,i2] of s1,
     array3 = array [i .. j] of boolean,
     array4 = array [1 .. 10] of array3
```

~~The array types, "array1" and "array2" are equivalent. The "array3" type may be of variable bounds because its index range cannot be determined until run-time elaboration of the declaration. Similarly for the "array4" type, since its component type is "array3".~~

4.3.4 RECORD TYPE

In a record structure, the components are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing the program), a record 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. Again, the time needed to access a selected component does not depend on the selector, and the record is like an array, a random-access structure.

A 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.

```
<record type> ::= [<packing>] <record type identifier>
                 |[<packing>] <record spec>
<record type identifier> ::= <identifier>
<record spec> ::= record <field list> recend
<field list> ::= [<fixed fields>,] <shifty field>
               | <fixed fields>
<fixed fields> ::= <fixed field> { ,<fixed field> }
<fixed field> ::= <field selectors> : ~~[<alignment>]~~ <fixed type>
<shifty field> ::= <variable bound field> ⋈
                 | <variant field>
<variable bound field> ::= <field selector>:~~[<alignment>]~~ <variable bound type> ⋈
<variant field> ::= case <tag field spec> of <variations> casend
<tag field spec> ::= <tag field selector> : <tag field type>
<tag field type> ::= <scalar type>
<tag field selector> ::= <identifier>
<variations> ::= <variation> { <variation>}
<variation> ::= = <selection values> = <variant>
<selection values> ::= <selection value> { <selection value>}
```

⋈ Release 1 does not support variable bound fields.

```
<selection value> ::= <constant scalar ~~expression>~~ ~~{.. <constant scalar expression>}~~
<variant> ::= [<fixed fields>] <variant field>
            | <fixed fields>
<field selectors> ::= <field selector>{,<field selector>}
<field selector> ::= <identifier>
```

A record type represents a structure consisting of a fixed number of components called fields, which may be of different types and are identified by field selectors, which are unique within any one variant and the preceding fixed fields. Multiple field selectors provide a concise notation for specifying fields having the same ~~alignment and~~ type. A record type whose last field is of variable bound type is called a variable bound record type; one whose last field is a variant field is called a variant record type, which may never be of variable bound type.

A variant field is distinguished by an explicit tag field which represents a set of selection values that maps onto the variants in a many-to-one manner.

Two record types are equivalent if they have the same packing, the same number of fields, and identical field selectors and equivalent types for corresponding fields. Two variant fields are of equivalent types if they have identical tag field selectors and equivalent tag field types, and if variants having identical field selectors and equivalent types are selected by the same selection values. The type of a variable bound field is determined when the declaration is elaborated.*

* In ISWL, two types are equivalent only if they represent the same instance of a type definition.

Example:

```
type

      date ≐ record day:  1 .. 31,
                    month:  string(4) of char,
                    year:  1900 .. 2100
             recend,
      status = record age:  6 .. 66,
                    married, sex:  boolean,
              recend,
      red_book = record name:  string (3) of char,
                       status:  status,
                       scores:  array[0 .. n] of date
                  recend,
      shape = (triangle, rectangle, circle),
      angle = -180 .. 180,
      figure = record x, y, area:  real,
               case s:  shape of
               = triangle = side: real, inclination, angle1, angle2: angle
               = rectangle = side1, side2: real, skew, angle3: angle
               = circle = diameter: real
               casend
               recend
```

"Red book" type may be of variable bounds type by virtue of the index range of "scores", while "figure" type is of variant record type.

~~4.3.5 UNION TYPE~~

~~Union type represents a finite set of selectable, non-equivalent types.~~

```
~~<union type> ::= union (<type list>)~~
~~<type list> ::= <type>{,<type>}~~
```

~~Union types permit one to define procedures whose parameters can be of more than one type and provide a restrictive, but more sanitary, alternative to variant record types.~~

~~Two union types are equivalent if their type lists can be re-ordered so that corresponding types are equivalent.~~

~~Example:~~

```
~~type param = union (real, int)~~
```

4.4 STORAGE TYPES *

Storage types represent structures to which other variables may be added, deleted, and referenced under explicit program control (c.f., STORAGE MANAGEMENT STATEMENTS, 10.4). They are, in addition, the only SWL types that can be used to construct relative pointers (c.f., Pointer Types, 4.2.3).

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

Storage types may be of variable bound type (c.f., Fixed and Variable Bound Types, 4.2.0).

4.4.1 STACK TYPE

```
<stack type> ::= stack [<stack size>] of <type>
<stack size> ::= <integer expression>
```

A stack type represents a collection of up to "stack size" components (of the same type) accessed via a "last in-first out" discipline.

The "top" component of a stack (a variable of stack type) can be referenced by using the stack's identifier as a pointer.

* No storage types may be declared in Release 1. The only storage type supported is the default heap.

4.4.2 SEQUENCE TYPE

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

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

<span> ::= [<integer expression> rep]<type>

A sequence type represents a storage structure whose components are referenced by a sequential accessing discipline.

Example:

seq (100 rep integer, 30 rep array [1 .. 30] of char)

4.4.3 HEAP TYPE

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

A heap type represent: a structure whose components can be explicitly allocated and freed.

4.4.4 SEQUENCE AND HEAP SPACE

A space attribute of the general form

n1 rep type1, n2 rep 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. The space attribute has no other connotations whatever except those that may exist in the mind of the programmer.

N.B. No storage types may be declared in Release 1.

4.5 ADAPTABLE TYPES ⋈

Adaptable types are structural skeletons of structured and storage types containing one or more indefinite bounds, indicated by an asterisk. They may be used solely to define formal parameters of procedures (c.f., Procedure Type, 4.6.2) and adaptable pointers (c.f., Pointer Type, 4.2.3), the latter providing a mechanism for referencing fixed instances of adaptable types.

```
<adaptable type> ::=  <adaptable structured type>
                    | <adaptable storage type>
<adaptable structured type> ::=  <adaptable string>
                               | <adaptable array>
                               | <adaptable record>
<adaptable storage type> ::=  <adaptable stack>
                            | <adaptable sequence>
                            | <adaptable heap>
```

4.5.1 ADAPTABLE STRING

```
<adaptable string> ::= string(*) of <character type>
                     | <adaptable string identifier>
<adaptable string identifier> ::= <identifier>
```

⋈ Release 1 does not support adaptable types.

4.5.2 ADAPTABLE ARRAY

<adaptable array> ::= [<packing>] <adaptable array identifier>
                    | [<packing>] <adaptable array spec>

<adaptable array identifier> ::= <identifier>

<adaptable array spec> ::= array [<starred list>] of <type>

<starred list> ::= <star or index> {,<star or index>}

<star or index> ::= * : <scalar type> | <index> | *

An asterisk (*) without a scalar type indicates an adaptable bound of integer type.

4.5.3 ADAPTABLE RECORD

<adaptable record> ::= [<packing>]<adaptable record identifier>
                     | [<packing>]<adaptable record spec>

<adaptable record identifier> ::= <identifier>

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

4.5.4 ADAPTABLE STACK

<adaptable stack> ::= <adaptable stack identifier>
                    | stack[*] of <type>

<adaptable stack identifier> ::= <identifier>

N.B. Release 1 does not support adaptable types.

4.5.5 ADAPTABLE SEQUENCE

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

4.5.6 ADAPTABLE HEAP

```
<adaptable heap>::=  <adaptable heap identifier>
                   | heap(*)
<adaptable heap identifier> ::= <identifier>
```

N.B. Release 1 does not support adaptable types.

4.6 FORMAL TYPES [⋈]

```
<formal type> ::=  <label type>
                 | <procedure type>
                 ~~| <coprocess type>~~
```

Formal types may be used solely to define <u>formal reference parameters</u> (c.f., below) and <u>formal pointers</u> (c.f., Pointer Type, 4.2.3). See section 8.0 for semantics.

4.6.1 LABEL TYPE

```
<label type> ::= label
```

4.6.2 PROCEDURE TYPE

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

```
<procedure type> ::=  <procedure type identifier>
                    | proc [<parameter list>] [<return type>].
<procedure type identifier> ::= <identifier>
<parameter list> ::= (<param segment> {;<param segment>})
<param segment> ::= <reference params> | <value params>
<reference params> ::= ref <formal param list> :~~[read]~~<ref type>
<value params> ::= val <formal param list>     :~~[read]~~<val type>
```

[⋈] Release 1 does not support formal types.

<formal param list> ::= <identifier list>

<ref type> ::= <SWL type>

<val type> ::= <type> | <adaptable type> ⋈

~~<method> ::= ref | val~~

<return type> ::= <basic type>

Val type is further restricted to exclude the so-called non-value types: storage types, arrays of non-value types, and records containing a field of a non-value type.

Two procedure types are equivalent if corresponding param segments have the same number of formal parameters, identical methods and equivalent types, and if their return types are equivalent.⋈⋈~~The read access attribute (c.f., 7.1.1.1) defines a read-only parameter~~.

~~4.6.3 COPROCESS TYPE~~

~~<coprocess type> ::= coproc~~

⋈ Release 1 does not support adaptable types.

⋈⋈ In ISWL, two types are equivalent only if they represent the same instance of a type definition.

N.B. Release 1 does not support formal types.

4.7 FILE TYPE ⋈

A file type represents a source and/or sink of data whose components, like those of a sequence, are handled by a sequential accessing discipline. Although variables of file type may be declared, their identifiers are treated as formal (or so-called, 'logical') file-identifiers; the actual file-identifiers are outside the lexical scope of any SWL program, and their association with formal identifiers cannot be expressed in SWL (e.g., one cannot 'open' a file in SWL). All SWL file variables have the de-facto static attribute.

```
<file type> ::=  <file type identifier>
               | file [(<file attribute>)]
<file type identifier> ::=  <identifier>
<file attribute> ::= binary | text
```

A binary file's components are SWL variables; a text file's components are of string type, and are called lines. If no attribute is specified, the attribute text is assumed.

File components are referenced by so-called input-output statements (c.f., 10.5).

⋈ Release 1 does not support file types.

4.8 PACKING AND ALIGNMENT

<packing attributes> ::= packed ~~| unpacked~~

~~<alignment> ::= aligned~~

A packed structure will generally require less space at the cost of greater overhead associated with access to its components. If a packing attribute is unspecified then the structure is assumed to be unpacked. ~~An inner structure inherits the packing of its immediately containing structure unless the packing of the inner structure is explicitly specified.~~

Unpacked structures and their components are always aligned. Packed structures are also aligned unless they are components of a packed structure, but their components are not. ~~unless they are explicitly aligned.~~

~~The attributes packed, unpacked, and crammed (c.f., Crammed Types, 13.1.2) cannot be applied to types that are explicitly packed, unpacked, or crammed.~~

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS

5.1 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; value constructors are used to denote instances of values of set, array, and record types. There are two kinds of value constructors: definite value constructors, which include specific type identification; and indefinite value constructors, whose type must be determined contextually.[⋈]

5.1.1 CONSTANTS AND CONSTANT DECLARATIONS

<constant declaration> ::= const <constant spec list>

<constant spec list> ::= <constant spec> { ,<constant spec>}

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

<constant identifier ~~list~~> ::= <identifier ~~list~~>

A constant spec associates one ~~or more~~ identifier~~s~~ with the value of the constant. ~~expression. A constant expression is an expression whose factors are either constants or parenthesized constant expressions (c.f., Constants, 3.2.4, and Expressions, 9.0).~~

⋈ ISWL supports only definite value constructors for sets, and indefinite value constructors for array initialization.

5.1.2 DEFINITE VALUE CONSTRUCTORS

```
<definite value constructor> ::= $<value type id> [<value elements>]
<value type id> ::=  <set type identifier>
                     ~~| <array type identifier>~~
                     ~~| <record type identifier>~~
<value elements> ::= <value element>{,<value element>}
<value element> ::= [<rep spec>] <expression>
                     ~~| [<rep spec>] <indefinite value constructor>~~
<rep spec> ::= <positive integer expression> rep
```

Identifiers for definite value constructors are obtained by prefixing the "target 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 structure type. Definite value constructors may be used wherever an expression can be used.

5.1.3 INDEFINITE VALUE CONSTRUCTORS

```
<indefinite value constructor> ::= [<value elements>]
```

Indefinite value constructors can be used only where their their type is explicitly indicated by the context in which they occur: as arguments of conversion functions (c.f., Section 5.2), as elements of definite and indefinite value constructors, and for the initialization of variables (c.f., Section 6.0). They may be a set, array or record depending on their context. ISWL supports indefinite value constructors for array initialization only, and in this case the value elements must be constants.

~~Example:~~

~~For the types defined by~~

```
        type color = (red, green, blue),
                  S = string (3) of char,
                  A = array [1 .. 20] of integer,
                 R1 = record t : array [1 .. 3] of boolean,
                             s : S
                      recend,
                 R2 = record F1 : set of color,
                             F2 : S,
                             F3 : A,
                             F4 : R1
                      recend;
```

~~instances of definite value constructors for the types R1 and R2 follow, with their fine structure displayed.~~

```
        $R1 [[3 rep true] , 'SBC']
             ︸︸︸︸︸︸︸︸︸   ︸︸︸︸︸
                  |            └───<string constant> for field s
                  └───<indefinite value constructor> for field t
```

~~$R2 [[red, blue], 'CBS', [20 rep 2], [[3 rep false], 'BCS']]~~

~~<indefinite value constructor> for F4~~

~~<indefinite value constructor> for F3~~

~~<string constant> for F3~~

~~<indefinite value constructor> for F1~~

~~Each of the constants used in the above examples could have been replaced by expressions that evaluate to the required types.~~

5.2 TYPE CONFORMITY AND VALUE CONVERSION

The operations of assignment and comparison (and most binary operations) are defined only for operands of equivalent types. This requirement is relaxed only to permit values of a subrange type and values of its parent type to enter into the same operation. When it is necessary to operate on operands that do not meet the strict requirements for type equivalence, the conversion functions described below must be used. These map values of a "source" type into values of a "target" type and are defined only for so-called _conformable_ source and target types.

5.2.1 TYPE CONFORMITY

Type conformity is a weak form of equivalence that does not require identical "computational" attributes, bounds, packing, alignment or field selectors. Unlike equivalence, conformity is not a transitive concept.

Integer types and _character_ types conform.
Integer types and _real_ types conform.
Integer types and _ordinal_ types conform.
String types are mutually conformable.
Array types conform if they have the same dimension, their corresponding index ranges span the same number of elements and their component types conform.
Record types conform if they have the same number of fields and corresponding fields conform.
String types, and one-dimensional _array_ types with character-type components, conform.

5.2.2 TYPE CONVERSION FUNCTIONS

Identifiers for conversion functions are obtained by prefixing the target type identifier with a dollar sign. The function so identified will then accept as an argument values that are in type conformity with the target type.

5.2.2.1 Primitive Conversions

These consist of the "pre-defined" functions (c.f., Standard Functions, 11.2).

```
        $integer (<real expression or char expression or ordinal expression>)
        $real (<integer expression>)
        $char (<integer expression>)
        $string (<length>, <string expression>[,<char expression>])
```

and the "definable" functions (c.f., below)

```
        $<ordinal type identifier> (<integer expression>)
        $<string type identifier> (<string expression> [,<char expression>])
```

5.2.2.1.1 Pre-defined Primitive Conversions

Conversions between the basic types are the conventional ones and are defined in section 11.2. In conversions between string types, the source string is converted to match the specified length either by truncation (on the right) or by appending (on the right) the required number of so-called "fill" characters. In the "long" form of the string conversion function, the fill character is explicitly specified by the last parameter; in the "short" form it is implicitly specified to be the space character.

5.2.2.1.2 Definable Primitive Conversions

Conversions to ordinal type return the value whose ordinal number is the value of the integer expression used as argument.

String-to-string type conversions are analagous to the pre-defined string-conversion function, with the length being specified by the length associated with the target type.

5.2.2.2 Structured Conversions

Conversions between strings and one-dimensional arrays of characters are analagous to string-to-string conversions. No other structured conversions are supported by ISWL.

~~Array-to-array and record-to-record conversions are defined recursively in terms of component conversions. When a structured conversion involves string component conversion, the fill character specified in, or implied by, the functional form will be used whenever filling is required. The two forms are identical to the two forms of definable string-to-string conversions:~~

~~$<array type identifier> (<array expression>[,<char expression>])~~

~~$<record type identifier> (<record expression>[,<char expression>])~~

~~<array expression> ::= <expression>~~

~~| <indefinite value constructor>~~

~~<record expression> ::= <expression>~~

~~| <indefinite value constructor>~~

```
type    A1 = array [1 .. 10] of s1,
         s1 = string (20) of char,
        A2 = array [11 .. 20] of s2,
         s2 = string(10) of char,
        R1 = record
                    i : integer,
                    j : real,
                   A : A1
              recend,
        R2 = record
                   alpha : real,
                   beta : real,
                   gamma : A2
              recend,
        R3 = packed  R2;
```

~~The two array types conform, the two string types conform, and the three record types conform.~~

6.0 VARIABLES, ATTRIBUTES AND SEGMENTS

6.1 VARIABLE DECLARATION

<variable declaration> ::= var <variable specs>

<variable specs> ::= <variable spec> {,<variable spec>}

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

<variable identifiers> ::= <variable identifier> {,<variable identifier>}

<variable identifier> ::= <identifier>

A variable spec introduces a new variable in terms of the identifier that denotes the variable, a type, an optional set of so-called attributes and an optional value initialization (c.f., 6.3).

6.2 ATTRIBUTES

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

<attribute> ::= ~~<access attribute>~~

| <storage attribute>

| <scope attribute>

~~6.2.1 ACCESS ATTRIBUTE~~

~~<access attribute> ::= read | write | execute~~

~~A variable can be declared sans access attributes or with the read attribute; no other combinations are allowed. In the former case, the variable can be used freely in expressions (c.f., Section 9.0) and as an object for assignment (c.f., Section 10.1);~~

~~In the latter case, the variable may be initialized, may not be an object for assignment, and may be used as an actual parameter only if the corresponding formal parameter is a val parameter or a read-attributed ref parameter (c.f., Section 4.6.2). Access to such so-called "read-only" variables may or may not be completely enforceable; thus, the results of attempting to alter a read-only variable (except by direct assignments) may be undefined.~~

6.2.2 STORAGE ATTRIBUTES [*]

<storage attribute> ::= static ~~| <segment identifier>~~

Storage attributes specify when storage for a variable is to be allocated (and initialized if necessary) and freed. If neither storage attributes nor scope attributes (see below) are specified, allocation and initialization occur automatically each time the variable declaration is elaborated (on entry to the block containing the declaration), and freeing occurs automatically on each exit from that block (c.f., Section 7.0). Variables so treated are called automatic variables. If a storage attribute is specified, then allocation and initialization occur once and only once – at a time no later than initial entry to the block containing the declaration, and storage is not freed on exits from that block. ~~When the storage attribute is a segment identifier (c.f., Section 6.5), then storage is allocated within the specified segment. Segments have the de-facto static attribute.~~

6.2.3 SCOPE ATTRIBUTES

<scope attribute> ::= xdcl | xref ~~| external~~

[*] Release 1 does not support static storage.

Variable identifiers are used in varable 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 block in which it occurs, and a so-called block-structuring regimen is used (c.f., Section 7.2). 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. ~~External variables are introduced to permit SWL programs to be interfaced with programs written in other languages; they may be referenced whenever and wherever their spec appears.~~ Neither xref nor external variables can be initialized, and each carries the de-facto static storage attribute.

6.3 INITIALIZATION

```
<initialization> ::=  := ~~<expression>~~ <constant>
                    | := <indefinite value constructor>
```

Since initialization is an "allocation-time" assignment to the variable, the initialization expression must satisfy the requirements of the assignment statement (c.f., 10.1).

~~An asterisk, "*", can be used to denote an uninitialized element of a value constructor used for initialization.~~

~~Non-constant initialization expressions are evaluated when the variable declaration is elaborated – on entry to the block containing the declaration. Initialization of static variables is restricted to constant expressions or value constructors all of whose fields are constants.~~

Examples of initialization are used throughout the remainder of this section in the explanation of variable references.

ISWL only supports initialization of static variables which are scalars, strings, or unpacked arrays. Initialization is restricted to constants or value constructors all of whose fields are constants.

6.4 VARIABLE REFERENCES

```
<variable> ::= <variable reference>
<variable reference> ::=   <variable identifier>
                         | <pointer reference>^
                         | <substring reference>   ⋈
                         | <subscripted reference>
                         | <field reference>
```

6.4.1 POINTER REFERENCES

```
<pointer reference> ::= <pointer variable> | <function designator>
<pointer variable> ::= <variable>
```

Whenever a variable reference denotes a variable of pointer type it is referred to as a <u>pointer reference</u> 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 procedures.

⋈ Release 1 does not support substring references.

6.4.2 SUBSTRING REFERENCES [⋈]

```
<substring reference> ::= <string variable> (<substring spec>)
<string variable> ::= <variable>
<substring spec> ::= <first char> [,<substring length>]
<first char> ::= <positive integer expression>
<substring length> ::=  <positive integer expression>
                     | *
```

Values of string variables (of length n) are ordered n-tuples of character values. Substring references denote ordered sub-tuples of string variables. If "S" denotes a string variable (of length, say, n) then: "S(i)" denotes the i-th character of S; S(i,k)" denotes the sub-tuple of S consisting of the i-th through the (i+k-1)-th character of S; "S(i,*)" is equivalent to "S(i,n-i+1)".

For purposes of type equivalency, "S(i,k)" denotes a value of type string(k).

Example:

If a string variable is declared by

```
var S : string(6) of char := '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.
```

⋈ Release 1 does not support substring references.

6.4.3 SUBSCRIPTED REFERENCE

```
<subscripted reference> ::= <array variable> [<subscripts>]
<array variable> ::= <variable>
<subscripts> ::= <subscript>  { ,<subscript> }
<subscript> ::= <scalar expression>
```

A subscripted reference denotes a component of an array variable, whose value type is the component type of the array variable. Subscript types must be equivalent to the corresponding index types specified for the array variable. However, for purposes of computational equivalency, values of a subrange type and values of the parent type are treated as being of equivalent type (the parent type).

Example:

If an array variable is specified by

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

and an integer variable is specified by

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

then the following relations hold

```
A[i] = 5
A[i-1] = 4
  :
A[i-4] = 1
```

6.4.4 FIELD REFERENCES

```
<field reference> ::= <record variable>.<field selector>
<record variable> ::= <variable>
```

A field reference denotes a field of a record variable. Since field selectors are unique only within the scope of their parent record type, the record variable must be specified. The field denoted by a field reference may be of record type, in which case

```
<record variable>.<field selector>.<field selector>.
```

becomes a valid field reference.

The field identifiers within a variant are available as field selectors only within the constituent statement list of a variant case statement (c.f., Section 10.2.8).

Example:

For the record variable defined by

```
var R : record age : 6 .. 66,
               married, sex : boolean,
               date : record day : 1 .. 31,
                             month : 1 .. 12,
                             year : 70 .. 80
                      recend,
        recend
        := [23,false,true,[3,5,73]]
```

the following relations hold

```
R.age = 23
R.date.year = 73 .
```

~~6.5 SEGMENTS AND SEGMENT DECLARATIONS~~

~~<segment declaration> ::= segment <segments> ,<segments>~~

~~<segments> ::= <segment identifiers> : [[<access attributes>]]~~

~~<segment identifiers> ::= <segment identifier> { ,<segment identifier> }~~

~~<segment identifier> ::= <identifier>~~

~~<access attributes> ::= <access attribute> { ,<access attribute> }~~

~~A segment is a static storage area for specified variables and procedures. The access attributes of variables and procedures declared to be in a particular segment must be a subset of that segment's access attributes. The combinations of segment access attributes to be supported will be implementation dependent, but will include [read], [read, write] and [execute].~~

~~Example:~~

```
segment symbol_info : [read,write];
var     name_table : [symbol_info] array [alot] of name ,
        attribute_table : [symbol_info] array [alot] of attr ,
        keywords : [symbol_info,read ]array [32] of name ;
```

7.0 BLOCKS, MODULES, AND COMPILATION UNIT

7.1 DECLARATIONS

Through the use of a declaration an identifier can be declared as a symbol with specific declared attributes. The range of references over which the identifier retains its declared meaning is known as the "scope" of the identifier.

```
<declaration list> ::= <declaration>{;<declaration>}; | <empty>
<declaration> ::= ~~<micro declaration>~~        ~~(c.f., 12.0)~~
                  ~~| <segment declaration>~~    ~~(c.f., 5.0)~~
                  | <constant declaration>       (c.f., 5.1)
                  | <type declaration>           (c.f., 4.1)
                  | <variable declaration>       (c.f., 6.1)
                  ~~| <module declaration>~~     ~~(c.f., 7.3)~~
                  | <label declaration>          (c.f., 8.2)
                  | <procedure declaration>      (c.f., 8.1)
```

7.2 BLOCKS ⋈

Unless further limited, the scope of an identifier is the block in which the identifier is declared. Thus, the symbol is known within the block and the block's inner blocks, but is unknown outside the block.

```
<block> ::= <begin statement>          (c.f., 10.2.1) ⋈
          | <procedure declaration>    (c.f., 8.0)
```

Example:

```
var choices : set of 'B' .. 'Y',
   last, result : union (boolean, 'B' .. 'Y');
begin
  var last : [static] 'A' .. 'Z';
  while last < 'Z' do
    last := #succ (last);
    if last in choices then
      result := last
    orif last = 'Z' then
      last := 'A';
      result := false
    ifend
  whilend
end;
if not (boolean :: result) then
  last := result
ifend
```

⋈ Release 1 does not support begin blocks.

~~7.3 MODULES~~

~~A module is a shield around a set of declarations.~~

~~<module declaration> := module [<module identifier>] [(<prongs>)];~~
~~<declaration list>~~
~~modend [<module identifier>]~~

~~An identifier declared within a module cannot be referenced from without the module unless the identifier is declared as a prong.~~

~~<prongs> ::= <identifier list>~~
~~<module identifier> ::= <identifier>~~

~~Declaring an identifier as a prong makes that identifier known immediately outside the module.~~

~~Example:~~

~~module (upper_case);~~
~~var mine : set of upper_case;~~
~~type upper_case = 'A' .. 'Z'~~
~~modend~~

~~is equivalent to~~

~~type upper_case = 'A' .. 'Z';~~
~~module;~~
~~var mine : set of upper_case~~
~~modend~~

~~Note that a module is not a block and that the declaration within the module are evaluated when the declarations of the block containing the module are evaluated.~~

7.4 GLOBAL VARIABLES

A variable declared with the xdcl, xref, ~~or external~~ attribute (c.f., 6.2.3) is a static global variable in the sense that it can be referenced by any other program that declares it.

7.5 COMPILATION UNIT

A compilation unit is the basic unit of input that can be compiled.

```
<compilation unit> ::= ~~<module declaration>~~ module [<module identifier>]
                                                <declaration list>
                                                modend [<module identifier>]
```

Any variables declared within the declaration list ~~outermost module~~ but not in a procedure declaration will implicitly have the static attribute. ~~Any variable that is declared as a prong in the outermost module will implicitly be given the xdcl attribute unless it was declared with the xref or external attribute.~~

7.6 FIELD SELECTORS

The scope of an identifier that is a field selector is the record definition itself (c.f., 4.3.4, Record Type).

8.0 PROCS, ~~COPROCS,~~ AND LABELS

A procedure 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 can return a value of some basic type, in which case it is referred to as a function and is invoked as a factor in an expression. If a procedure returns no value it is invoked by a procedure call statement.~~ or a coprocess create statement.~~

The value of a function is the value last assigned to its procedure identifier before returning (either by falling through the procend or by an explicit return statement).

A procedure call statement causes the execution of the constituent declarations and statement lists of the procedure after substituting the actual parameters of the call for the formal parameters of the declaration. Control returns to the next statement in line.

~~The create statement creates the necessary environment for the execution of a procedure as a coprocess. A coprocess is a separate synchronous process. Instead of the entire procedure being executed and then returning in line, coprocesses allow partial execution of a set of procedures with the single thread of control being passed back and forth amongst them through the resume statement. Subsequent resumption of a coprocess causes execution to commence with the successor of the last executed resume statement of the coprocess. If a coprocess has been created but not resumed, then execution of a resume statement designating that coprocess causes execution to commence at the constituent declaration list of the procedure used to create the coprocess.~~

8.1 PROCEDURE DECLARATIONS

```
<procedure declaration> ::= proc [xref] <procedure identifier> [<parameter list>]
                            [<return type>]
                        | proc [[<proc attributes>]] <procedure identifier>
                            [<parameter list>][<return type>];
                            <declaration list><statement list>procend [<procedure identifier>]
<procedure identifier> ::= <identifier>
<function identifier> ::= <procedure identifier>
```

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

The second form declares the procedure identifier to be a procedure of the type specified by its parameter list and return type, and associates the identifier with the constituent declaration list and statement list of the declaration.

The type of the procedure is elaborated on entry to the block in which it is declared and remains fixed throughout the execution of that block, i.e., all variable bounds, lengths, or sizes are evaluated at that time.

Outermost level procedures, i.e., those whose declarations are not contained in another procedure, must therefore have a fixed type determined at compile time. Thus none of its parameters may be of a variable bound type. Note that this restriction holds with respect to the xref form of declaration since by definition it must refer to an outermost level procedure (Section 8.1.1, PROC ATTRIBUTES).

8.1.1 PROC ATTRIBUTES

Proc attributes are essentially extra-linguistic features in that they have an effect on the output produced by the compiler rather than an effect on the meaning of the program.

```
<proc attributes> ::= <proc attribute> ~~{ <proc attribute> }~~
<proc attribute> ::= xdcl ~~| repdep | <segment identifier>~~
```

The attribute xdcl may only be used on a procedure declared at the outermost level, i.e., not contained in another procedure. It specifies that the procedure should be made referenceable from other units of compilation which have a declaration for the same procedure identifier with the xref attribute.

~~The attribute repdep specifies that the procedure is potentially representation dependent and gives permission for the use of those portions of the language that are representation dependent (see Chapter 13).~~

~~The attribute "segment identifier" specifies that the code produced by the compiler for the body of the procedure should be allocated to the named segment along with other code and data carrying the same segment identifier.~~

8.1.2 PARAMETER LIST

Variables that are referenced but not declared in the body of a procedure follow normal scope rules, i.e., the references are bound to the declaration environment of the procedure. A parameter list is a set of variable declarations which provides a mechanism for the binding of references to the procedure call environment. This is accomplished by providing the procedure with a set of values and variables – so called actual parameters – at the point of call.

```
<parameter list> ::= (<parameter segment> { ; <parameter segment> })
<parameter segment> ::= <method> <parameters> { , <parameters> }
<method> ::= val | ref
<parameters> ::= <parameter> { , <parameter> } : ~~[[read]]~~ <SWL type>
<parameter> ::= <identifier>
```

Two methods of passing parameters are provided – call-by-value, designated by val, and call-by-reference, designated by ref.

A call-by-value parameter results in the creation of a variable of the specified type local to the body of the procedure. The value of the corresponding actual parameter is assigned to this variable at the time of the procedure call. If the formal parameter is an adaptable array, string, or record then the variable thus created is an array, string, or record of the same size and shape as the corresponding actual parameter.

The type of a formal call-by-value parameter may be any data type or adaptable type except for the so-called non-value types. The non-value types are: file, stack, heap and sequence, arrays of non-value types, and records containing a field of a non-value type.

N.B. Release 1 does not support variable bound types, adaptable types, or any of the non-value types.

A call-by-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 call-by-reference parameter may be any SWL type (including the non-value types, and the formal types – label, procedure and coprocess).

~~The read attribute applied to either kind of parameter prohibits explicit assignments to that parameter or any component of it.~~

The procedure type is elaborated on entry to the block in which it is declared, and remains fixed throughout the execution of that block, i.e., all variable bounds, lengths, or sizes occurring in the type of the parameters are evaluated once on entry to the block, and remain fixed for all calls on the procedure within that block.

8.1.3 FUNCTIONS AND RETURN TYPE

A procedure may return a value of a specified type, in which case it is referred to as a function. A function is activated by a function designator (see Factors in Chapter 9) which is a component of an expression. The function is given a value by assigning to its procedure identifier. The type of the value returned is specified by the return type.

    <return type> ::= <basic type>

Examples:

```
proc  GCD (val m , n : integer; ref 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 / 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
```

8.2 LABEL DECLARATIONS

Label declarations serve to define those labels of the block which may be assigned to a pointer to label variable, passed as a parameter to a procedure or function, or serve as the destination of a goto exit statement which crosses a block or procedure boundary (see 10.3.8, GOTO STATEMENTS).

```
<label declaration> ::= label <label> { , <label> }
<label> ::= <identifier>
```

All labels in the list must also appear in the block labeling a statement which is not contained within a nested block (see 10.0, STATEMENTS).

~~Note that only those labels which are assigned, passed as a parameter, or are the destination of a non-local goto statement are required to be declared in a label declaration, but other labels of the block are permitted.~~

ISWL requires all labels to be declared.

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.

The rules of composition specify operator precedence according to five classes of operators. The not operator has ~~The type testing operators have~~ the highest precedence, ~~followed by the not operator,~~ followed by the so-called multiplying operators, then the so-called adding operators, and finally, with the lowest precedence, the relational operators. Sequences of operators of the same precedence are executed from left to right and the left operand of a dyadic operator is evaluated before the right operand. The rules of precedence are reflected by the following syntax:

~~<conformity> ::= <type identifier> <type test operator> <union variable>~~

~~| <pointer variable> <pointer type test operator> <union variable>~~

<factor> ::= ~~<conformity> |~~ <variable> | <constant>

| <definite value constructor> | ∧<variable> | ∧<label> ⋈

| ∧<procedure identifier> | <function designator> ⋈

| (<expression>) | <not operator> <factor>

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

<simple expression> ::= <term> | <sign> <term>

| <simple expression> <adding operator> <term>

<expression> ::= <simple expression>

| <simple expression> <relational operator> <simple expression>

~~<type identifier> ::= <identifier>~~

~~<union variable> ::= <variable>~~

<function designator> ::= <procedure reference> (<actual parameter>{,<actual parameter>})

~~| <procedure reference> ( )~~

⋈ Release 1 does not support ∧<label> or ∧<procedure identifier> .

<procedure reference> ::= <procedure identifier> | <pointer to procedure>∧ ⋈

<actual parameter> ::= <expression> | <procedure identifier> | <label> ⋈⋈

~~<type test operator> ::= ::~~

~~<pointer type test operator> ::= ::=~~

<not operator> ::= not

<multiplying operator> ::= * | / | mod | and

<sign> ::= + | -

<adding operator> ::= + | - | or | xor

<relational operator> ::= < | <= | > | >= | = | /= | in

Examples:

~~Conformities:~~	~~pint ::= basicvar~~
	~~real :: basicvar~~
Factors:	x
	15
	(x + y + z)
	f(x + y)
	$colorset [red, c, green]
	not p
	∧a [i, j]
Terms:	x * y
	i / 3
	p and q
	(x <= y) and (y < z)

⋈ Release 1 does not support <pointer to procedure>.

⋈⋈ Release 1 does not support procedure or label parameters.

9.1 EVALUATION OF FACTORS

~~The value of a conformity as a factor is the boolean value true if the type test is successful and false otherwise (see 9.2.1, Type Test Operators).~~

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 or real denoted by it in the specified radix system.

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

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

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

A constant identifier is replaced by the constant it denotes. If this in trun 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 definite value constructor is the value obtained from the values of its constituent expressions of type specified by its type identifier.

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

Simple expressions:	x + y
	-x
	hue1 _or_ hue2
	i * j + 1
	hue - $colorset [red, green]
Expressions:	x = 1
	p <= 2
	(i < j) = (j < k)
	c _in_ hue1

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

The value of an up-arrow followed by a label is the pointer to label value of type ∧label that designates the current instance of declaration of the label (see 10.0, STATEMENTS). ⋈

A function designator 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 designator is the value last assigned to the function identifier. The procedure reference must be to a procedure with a return type. The meaning of, and restrictions on, the actual parameters is the same as for the procedure call statement (see 10.3.1).

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 designator whose type is a subrange of some scalar type is that scalar type.

⋈ Release 1 does not support pointer to procedure or pointer to label values.

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.

~~9.2.1 TYPE TESTING OPERATORS~~

~~The type testing operators are used to determine the type of the value last assigned to a union variable. The type test operator (::) returns the boolean value true if the type identifier on the left specifies the same type as the type of the value of the union variable on the right, and false otherwise.~~

~~The pointer type test operator (::=) returns the boolean value true if the pointer variable on the left is of type pointer to the type of the value of the union variable on the right. If it is, then the pointer variable on the left is caused to designate the value of the union variable; otherwise the value is false and the pointer variable is assigned the value nil.~~

9.2.2 NOT OPERATOR

The not operator, not, applies to factors of type boolean and set. When applied to type boolean the meaning is negation - i.e., not true ≡ false and not false ≡ true. When applied to a set the meaning is set complement with respect to the base type - i.e., the set of all elements of the base type not contained in the specified set.

9.2.3 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	real integer	real integer
/	integer division for a, b, n positive integers a/b = n where n is the largest integer such that b * n <= a (-a)/b ≡ (a)/(-b) ≡ - (a/b), a/b ≡ (-a)/(-b)	integer	integer
	real division	real	real
mod	remainder function a mod b ≡ a - (a/b) * b	integer	integer
and	logical 'and' true and false ≡ false, true and true ≡ true false and false ≡ false, false and true ≡ false	boolean	boolean
	set intersection - the set consisting of elements common to the two sets.	set of type	set of type

9.2.4 SIGN OPERATORS

The + operator can be applied to integer and real types only. It denotes the identity operation and results in integer or real type respectively – i.e., $a \equiv +a$.

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

9.2.5 ADDING OPERATORS

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

Operator	Operation	Operands	Result
+	addition	integer real	integer real
-	subtraction	integer real	integer real
	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	set of type
or	logical 'or' true or true ≡ true, true or false ≡ true false or true ≡ true, false or false ≡ false	boolean	boolean
	set union - the set consisting of all elements of both sets.	set of type	set of type
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	set of type

9.2.6 RELATIONAL OPERATORS

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

9.2.6.1 Comparison of Scalars, Reals, and Strings.

All six comparison operators < (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, operands of type real, and operands of type string or string and char.

For operands of type integer or real they have their usual meaning.

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.

For operands of type string (n) or string (1) and char, comparison is defined in the following way:

If one of the operands is of type char it is converted to the string (1) value consisting of that character; otherwise the strings must be of the same length.

Let n be the length of the resulting string values a and b (n >= 1), and cp be any of the six comparison operators, then:

a op b iff a (1) op b (1)
or a (i) = b (i) for all i (1 < = < k)
and a (k) op b (k) (1 < k < = h)

9.2.6.2 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.

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.

9.2.6.3 Relations Involving Other Types

Certain types in the language cannot be compared. These are stacks, heaps, sequences, unions, variant records, arrays of non-comparable component types, and records containing a field of a non-comparable type. The remaining types (including pointers to non-comparable types) are comparable for equality (=) and inequality (/=).

Two arrays are equal if their types are the same (i.e., subscript bounds and component types are identical) and elements with corresponding subscript values are equal.

Two records are equal if they are of the same type and their corresponding fields are equal.

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

~~Two pointers to coproc are equal if they designate the same coprocess.~~

Two pointers to label are equal if they designate the same instance of definition of a label. ⋈

Pointers of other types are equal if they designate the same variable. For adaptable pointers this means that their instantaneous type (i.e., including bounds or lengths) must be the same as the pointer they are being compared with. ⋈⋈

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

⋈ Release 1 does not support pointers to procedure or pointers to label.

⋈⋈ Release 1 does not support adaptable types.

Operator	Operation	Left Operand	Right Operand	Result
< <= > >= = /=	- less than - less than or equal to - greater than - greater than equal to - equal to - not equal to	any scalar type T real string (n) string (1) char	T real string (n) char string (1)	boolean boolean boolean boolean boolean
in	set membership test	any scalar type T	set of T' where T' is T or a subrange of T	boolean
= /= <= >=	- identity - different - is contained in - contains	set of T where T is any scalar or subrange type	set of T	boolean
= /=	- equal to - not equal to	any comparable type T	T	boolean

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.

A statement may be labelled by preceding it by an identifier followed by a colon. This allows the statement to be explicitly referred to by other statements (e.g., goto, exit, cycle). Such a labelling of a statement constitutes the declaration of the identifier as a label, and hence the identifier must differ from all other identifiers declared in the same block.

If an identifier labels a statement of the constituent statement list of a procedure declaration (see Section 8.0) or a begin statement (see Section 10.2.1), then its scope is that procedure declaration or begin statement. If it labels a statement of one of the constituent statement lists of other structured statements (see section 10.2), then its scope is that statement list. Thus it is impossible to refer to a label contained within a procedure declaration or structured statement from outside that declaration or statement, or from other statement lists of the same structured statement. ⋈

A label may optionally follow a structured statement other than the repeat statement, in which case is must be identical to one of the labels labelling that statement. This is for checking purposes only, and does not affect the meaning of the program.

⋈ This paragraph in the SWL specification {7 Dec. 73} is incorrect and must be replaced.

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

<statement> ::= <unlabelled statement> | <label> : ~~<statement>~~<unlabelled
                                                                      statement>

<unlabelled statement> ::= <assignment statement>

                         | <structured statement> [<label>]

                         | <control statement>

                         | <storage management statement>

                         | <input-output statement>

<label> ::= <identifier>
```

Example:

```
check_range: if val < 0 then tagfld := 0
             orif val > bound then tagfld := bound
             else tagfld := val
             ifend check_range
```

10.1 ASSIGNMENT STATEMENTS

The assignment statement is used to replace the current value of a variable by a new value derived from an expression, or to define the value returned by a function designator.

```
<assignment statement> ::=  <variable> := <expression>
                          | <function identifier> := <expression>
```

The left part 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 not be of type file, sequence, stack or heap, nor may it be an array of such, nor a record containing a field of those types.

The variable on the left and the expression on the right must be of identical types except as noted below:

1. The type of the variable may be a subrange of the type of the expression. If the value of the expression is not a value of the subrange the program is in error.

2. ~~If the variable is a union variable, then the value of the expression may be any one of the types from which the union type was united. In this case, the type of the expression as well as its value is assigned.~~

3. If the left part is a string or substring designator of length 1, then the expression may be a char value, and if the left part is of type char then the expression may be a string value of length 1.

4. If the left part is a variable bound array then the expression must be an array with the same current values and types of subscript bounds, packing attribute, and component type. ⋈

5. If the left part is an adaptable pointer to type, the expression must be a pointer to one of the types to which the pointer can adapt. ⋈

Note that generally a pointer value has a finite lifetime (see Section 5) different from that of the pointer variable. Procedures, labels, and 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 popped. Attempts to reference them via a designator beyond their lifetime is a programming error and could lead to disastrous results.

Examples:

```
i := i + 1              ptr1 := ∧a [i]              myunion := true
a [i] := 15             a := b [i]                  myunion := x/15.0
ptr1 := ptr2            a[j]:= b[i,j]               errorbranch :=∧label1
buffer (i,20) := "ERROR IN DECLARATION"             strptr :=∧buffer(i)
```

⋈ Release 1 does not support variable bound or adaptable types.

10.2 STRUCTURED STATEMENTS

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

```
<structured statement> ::= <begin statement> ⋈
                         | <if statement> | <loop statement>
                         | <while statement> | <repeat statement>
                         | <for statement> | <case statement>
                         | <variant case statement> ~~| <conformity case statement>~~
```

10.2.1 BEGIN STATEMENTS ⋈

Begin statements are blocks, and constitute the scope of their constituent declarations. On entry to the begin statement all declarations are evaluated, and storage allocated for automatic variables. The statement list is then executed. On exit, either through completing execution of the last statement of the statement list or through an explicit transfer of control, all identifiers declared within the begin statement become inaccessible, and the values of automatic variables become undefined.

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

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

Example:

```
begin var temp:integer; temp := i, i := j; j := temp end
```

⋈ Release 1 does not support the begin statement.

10.2.2 IF STATEMENTS

The if statement provides for the execution of one of a set of statement lists depending on the values of Boolean expressions. The Boolean expressions following the if or orif symbols are evaluated in order from left to right until one is found whose value is true. The subsequent statement list is then executed.

If all Boolean expressions are false then either no statements or the statement list following the else symbol is executed.

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

```
<if statement> ::=  <alternative parts> ifend
                  | <alternative parts> else <statement list> ifend
<alternative parts> ::= if <expression> then <statement list>
                        {orif <expression> then <statement list>}
```

Examples:

```
if x < y then x := y ifend
if x <= 5 then z := y + 1; y := y + 5
orif x > 30 then z := y * y; y := z
orif x = 15 then z := y * z
else z := z * z; y := 2 * z + 15
ifend
```

Examples:

```
    while a[i] /= x do i := i + 1 whilend
    while i > 0 do
        if odd (i) then z := z * x ifend;
        i := i / 2;
        x := x * x
    whilend
```

10.2.5 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. The repeat statement

repeat S until e

is equivalent to

begin S; if e then else repeat S until e ifend end

The successor of the last statement of the constituent statement list of a repeat statement is the expression following until.

Example:

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

10.2.6 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> [~~by <increment>~~]
                | <initial value> downto <final value> [~~by <decrement>~~]
<control variable> ::= <variable>
<initial value> ::= <expression>
<final value> ::= <expression>
~~<increment> ::= <expression>~~
~~<decrement> ::= <expression>~~
```

The control variable, initial value, and final value ~~and increment or decrement~~ must all be of the same scalar type or subranges of the same type. ~~The control variable may not be a component of a packed, or crammed structure, and when the "by" option is used must be type integer or subrange thereof.~~ ISWL requires the control variable to be simple and either local or global.

The sequence of values assigned to the control variable for which the statement list is executed, is determined solely by the initial value and final value, ~~and increment or decrement.~~ Assignment to the control variable on a given iteration will cause its value to be changed for the remainder of that iteration, but its value will be reset to the next value of the sequence prior to the next iteration.

The initial value and final value, ~~and increment or decrement~~ 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 no assignment is made to the control variable by the statement list, and the statement is exited normally, then the value of the control variable is the final value.

A for statement of the form

```
    for w := i to n do S forend
```

is equivalent to

```
begin var control : ∧TYPE (w), temp, limit : TYPE (w)
control := ∧w; temp := i; limit := n;
if temp <= limit then
while temp < limit do control ∧:= temp, S; temp :=#succ(temp) whilend;
control ∧:= temp;
S;
ifend
end
```

where control, temp and limit are identifiers not appearing in the statement list S, and TYPE is a function returning the type of its argument (not available in SWL).

A for statement of the form

```
for w := i downto n do S forend
```

is equivalent to

```
begin var control : ∧TYPE(w), temp, limit : TYPE(w);
control := ∧w; temp := i, limit := n;
if temp >= limit then
while temp > limit do control∧ := temp; S; temp =#pred(temp) whilend;
control∧ := temp;
S;
ifend
end
```

A for statement of the form

```
for w := i to n by inc do S forend
```

is equivalent to

```
begin var control : ^ TYPE(w), limit, step, temp : integer;
        control := ^w; temp := i; limit := n; step := inc;
        while temp <= limit do
                control ^ := temp; S; temp := temp + step
        whilend
end
```

And a for statement of the form

```
for w:= i downto n by decr do S forend
```

is equivalent to

```
begin control : ^ TYPE(w), limit, step, temp : integer;
        control :=^w; temp := i; limit := n; step := decr;
        while temp >= limit do
                control ^ := temp; S; temp := temp - step
        whilend
end
```

The successor to the last statement of the constituent statement list of a for statement is the calculation of the next value of the temporary control variable.

Examples:

```
for i := 2 to 100 do if a[i] > max then max := a[i] ifend forend

for i := 1 to n do
     for j := n downto 1 do
          x := 0;
          for k := 1 to n do x := x + a[i, k] * b[k, j] forend;
          c[i, j] := x
     forend
forend

for c := red to blue do q(c) forend
```

10.2.7 CASE STATEMENTS

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

```
<case statement> ::= case <selector> of<cases> [else <statement list>] casend
<selector> ::= <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; otherwise 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).

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 := sin(x);
      = 2 =       x := cos(x);
      = 3 =       x := exp(x);
      = 4 =       x := ln(x)
      else        x := - x
casend
```

10.2.8 VARIANT CASE STATEMENTS

A variant case statement is used to allow access to the variant fields of a record according to the value of its tagfield.

```
<variant case statement> ::= case tag <tag selector> of <cases>
                                      [else <statement list>] casend
<tag selector> ::= <record variable> • <tagfield>
<record variable> ::= <variable>
<tagfield> ::= <identifier>
<cases> ::= <a case> {; <a case>}
<a case> ::= = <selection spec>{,<selection spec>} = <statement list>
<selection spec> ::= <constant scalar expression> [.. <constant scalar expression>]
```

Each selection specification list must identify a set of values which is identical to, or a subset of, the set of values which select a unique variant associated with the tag field in the corresponding record definition. The field identifiers of that variant are available as field selectors in the associated statement list.

If the value of the tagfield differs from the values of all selection specifications the statement list following else is selected and none of the field identifiers of the variants are made available as field selectors; if else was not specified the program is in error.

The successor of the selected statement list is the successor of the variant case statement.

Examples:

```
    type lextype = (basic, inconst, realconst, stringconst, identifier),
       symbol = record
                   case lex : lextype of
                   = basic = name : symbolid, class : operation
                   = inconst = value : integer, optimiz : boolean
                   = realconst = value : real
                   = stringconst = length : 1..255, stringbuf : ^string(*)
                   = identifier = identno : integer, decl : ^symbolentry
                   casend
                recend;
    var cursym : symbol, sign : boolean := false;
    L1 : insymbol;
    L2 : case tag cursym.lex of
       = basic = if cursym.symbolid = minus then sign := not sign; goto L1
                orif cursym.symbolid = plus then goto L1
                else error ('missing operand')
                ifend;
       = intconst = cursym.optimiz := (cursym.value < halfword) or pwr2 (cursym.value);
                if sign then sign := false; cursym.value := - cursym.value ifend;
       = realconst = if sign then sign := false; cursym.value := - cursym.value ifend;
       = stringconst = error ('string constant where arithmetic type expected');
       = identifier = cursym.decl := symbolsearch (cursym.identno);
                if cursym.decl^.typ /= constdecl then variable (cursym.decl )
                else cursym := cursym.decl^.value^; goto L2
                ifend
       casend
```

~~10.2.9 CONFORMITY CASE STATEMENTS~~

~~A conformity case statement selects for execution one of its component statement lists depending on the type of the value last assigned to a union variable.~~

~~<conformity case statement> ::=~~
~~case union <union variable> of <conformity cases> [else <statement list>] casend~~
~~<union variable> ::= <variable>~~
~~<conformity cases> ::= <a conformity case> {;<a conformity case>}~~
~~<a conformity case> ::= <type specification> = <statement list>~~
~~<type specification> ::= <pointer variable>~~
~~<pointer variable> ::= <variable>~~

~~Each type specification must be a pointer variable to one of the types of the union variable, and must be of a type different from all other type specifiers in the statement (i.e., type selection must be unique). If one of the type specifiers is a pointer to the type of value last assigned to the union variable, it will be assigned a pointer to that value and the associated statement list will be executed. Within the statement list the pointer followed by an up arrow may be used to refer to the value.~~

~~If none of the type specifiers match the type of the value of the union variable the statement list following else is executed; if the else part is omitted the program is in error.~~

~~The successor of the selected statement list is the successor of the conformity case statement.~~

~~Example:~~

```
proc format (ref u : union (integer, boolean), S : string(*));
var pint :∧ integer, pbool :∧ boolean;
case union u of
   = pint = stringrep (pint∧ , S, 12);
   = pbool = if pbool ∧ then S(1,6) := 'true__'
                          else S(1,6) := 'false_'
             ifend
casend
procend
```

10.3 CONTROL STATEMENTS

Control statements cause the creation or destruction of execution environments, the transfer of control to a different execution environment or to a different statement in the same environment, or both.

```
<control statement> ::=  <procedure call statement>
                         ~~| <create statement> | <destroy statement>~~
                         ~~| <resume statement>~~ | <cycle statement>
                         | <exit statement> | <return statement>
                         | <goto statement> | <empty statement>
```

10.3.1 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.

```
<procedure call statement> ::= <procedure reference> <actual parameter list>
<procedure reference> ::= <procedure identifier> | <pointer to procedure>^      *
<actual parameter list> ::=  (<actual parameter> {,<actual parameter>})
                          | <empty>
<actual parameter> ::=  <expression> | <procedure identifier>       **
                      | <label>
```

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 ordinal position in the formal parameter list.

* Release 1 does not support pointer to procedure.

** Release 1 does not support procedure or label parameters.

The corresponding actual and formal parameters must be of the same type except for the following:

1. If the formal parameter is call-by-value, the actual parameter may be any expression which could be assigned to a variable of the type of the formal. (See 10.1 ASSIGNMENT STATEMENTS)

2. If the formal parameter is an adaptable string, the actual parameter may be a string variable or substring designator of any length. If the formal were call-by-value then the actual may also be a string value. ⋈

3. If the formal parameter is an adaptable array, the actual parameter may be any array variable with the same packing attribute, number of dimensions, types of subscripts, and component type. If the formal were call-by-value then the actual may also be an array value with the same restrictions. ⋈

4. If the formal parameter is an adaptable record, the actual parameter may be any record whose type is the same except for the shifty field. The shifty field must be an adaptable string, array, or record and the conditions under 2, 3, or 4 hold with respect to it. ⋈

5. If the formal parameter is a call-by-ref procedure, then the actual parameter must be a procedure reference to a procedure with the same ordered list of parameter types and return type. ⋈⋈

6. If the formal parameter is a call-by-ref label, then the actual parameter must be a label reference. ⋈⋈

⋈ Release 1 does not support adaptable types.

⋈⋈ Release 1 does not support procedure or label parameters.

A call-by-value parameter causes the creation of a variable local to the procedure of the type of the formal parameter, and assigns the value of the actual parameter to it. If the formal is an adaptable array, string or record, the local variable is a fixed bound array, string or record of the same size and shape as the actual.

A call-by-ref 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 call-by-ref formal parameter may never be a component of a packed array or record.

<u>Examples:</u>

```
insymbol
transpose (a, n, m)
sum (fct, 0, 1000, x)
```

~~10.3.2 CREATE STATEMENT~~

~~The create statement causes the creation of a coprocess from the specified procedure, and establishes the environment (including parameter list) for the execution of that procedure as a coprocess. The identity of the created coprocess is assigned to the specified pointer to coproc. On completion of the create statement, a resume statement using the pointer to coproc would cause execution to be resumed at the constituent declaration list (if any) of the body of the procedure. The procedure specified in the create statement is designated the primary procedure of the coprocess. Exiting it through a normal return or a continue, exit, return or goto statement while the coprocess is still active is an error.~~

```
<create statement> ::= create (<pointer to coproc>, <procedure call statement>)
<pointer to coproc> ::= <variable>
```

Example:

```
create (nextsymbol, macro_expander (source-file))
```

10.3.3 DESTROY STATEMENT

The destroy statement causes the destruction of the coprocess specified by the pointer to coproc and sets the pointer to nil. Storage allocated to the coprocess is returned, and subsequent attempts to resume the coprocess or access variables local to it are in error. A destroy statement designating the coprocess in which it occurs is an error.

```
<destroy statement> ::= destroy (<pointer to coproc> {,<pointer to coproc>})
<pointer to coproc> ::= <variable>
```

Example:

```
destroy (nextsymbol)
```

10.3.4 RESUME STATEMENT

The resume statement causes execution of the current coprocess to be suspended, and execution to continue at the successor of the last executed resume statement of the specified coprocess. If the specified coprocess had just been created, execution resumes at its constituent declaration list.

~~A resume statement designating a destroyed coprocess or the coprocess in which it occurs is in error.~~

~~<resume statement> ::= resume (<coproc reference>)~~

~~<coproc reference> ::= <pointer to coproc>∧~~

~~Examples:~~

~~resume (nextsymbol∧)~~

~~resume (user[i]∧)~~

10.3.5 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, thus cycling it to its next iteration (if any).

<cycle statement> ::= cycle [<label>][when <expression>]

The label must label a repetitive statement (for, repeat, while or loop statement) which statically encompasses the cycle statement, i.e., the cycle statement must be within the scope of the label. If no label is specified then the continue statement must be a statement of the constituent statement list of a repetitive statement, and it is that repetitive statement that is cycled.

The expression following when must be a boolean expression. If the value of the expression is true, or the when clause does not occur, then execution continues at the successor of the last statement of the constituent statement list of the designated structured statement or procedure. Otherwise, execution continues at the successor of the cycle statement.

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

<u>Examples:</u>

```
for i = 1 to n do cycle when x < a[i]; x := a[i] forend
                  cycle outerloop when sum <= eps
```

10.3.6 EXIT STATEMENT

The exit statement causes execution to continue at the successor of the designated structured statement or procedure when the condition is true or non-existent. If no label or procedure is specified then execution continues at the successor of the immediately containing structured statement or procedure.

```
<exit statement>::= exit [<label or proc identifier>] [when <expression>]
<label or proc identifier> ::= <label> | <procedure identifier>
```

<u>Example:</u>

```
repeat exit when key = a[i]; i := i + 1 until i > n
```

10.3.7 RETURN STATEMENT

The return statement causes the current procedure to return when the expression is true or non-existent; i.e., the successor of a return statement is the successor of the last statement of the constituent statement list of the procedure or function in which it is embedded.

<return statement> ::= return [when <expression>]

Example:

return when next_term < epsilon

10.3.8 GOTO STATEMENT

<goto statement> ::= goto [exit] <label reference>

<label reference> ::= <label> | <pointer to label>^ ⋈

The goto statement names as its successor the labelled statement designed by the label or by the value of the pointer to label. ⋈

If the label reference is to a label outside the current block, then the form goto exit must be used, and the label must have been declared in a label declaration in the declaration list of its block; otherwise the form without exit is used. ⋈⋈

If the pointer to label designates a statement in a procedure that has already been exited, ~~or a statement in a coprocess other than the one in which the goto statement occurs,~~ then the goto statement is in error. ⋈

⋈ Release 1 does not support pointer to label.

⋈⋈ ISWL requires all labels to be declared in a label declaration.

Examples:

goto exit errexit

goto labelarray [symbol_number]∧

10.3.9 EMPTY STATEMENT

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

<empty statement> ::=

10.4 STORAGE MANAGEMENT STATEMENTS [*]

There are three storage types – stack, sequence, and heap – defined in the language, each with its own unique management and access characteristics. A variable of any of these types represents a structure to which other variables may be added, 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.

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

ALLOCATION DESIGNATOR

An allocation designator specifies the type of the variable to be managed by the storage management statements. It is either a pointer to tyep, in which case a variable of that type is designated, or it is an adaptable pointer variable followed by a type fixer which defines the adaptable bound, length or size, in which case it is a variable of the fixed type that is designated.

```
<allocation designator> ::= <pointer variable>
                          | <adaptable pointer to array> : [<bounds list>]
                          | <adaptable pointer to stack> : [<expression>]
                          | <adaptable pointer to record> : [<bounds list>]
                          | <adaptable pointer to string> : (<length>)
                          | <adaptable pointer to sequence> : <span>{,<span>})
                          | <adaptable pointer to heap> : (<span>{,<span>})   [**]

<span> ::= [<integer expression> rep] <type>
```

[*] Release 1 does not support stacks or sequences, and the only heap supported is the universal heap.

[**] Release 1 does not support adaptable types.

Examples:

procvector : [0 .. 10]
garbage : (35 rep rinds, 15 rep pits, grounds)

10.4.1 PUSH STATEMENT ⋈

The push statement causes a new element to be added to the specified stack. It is accessible through the stack variable followed by an up arrow.

<push statement> ::= push <stack variable>
<stack variable> ::= <variable>

Examples:

push operator_stack
"new top element is
operator_stack ∧"

10.4.2 POP STATEMENT ⋈

The pop statement causes the top element of the stack to be removed (i.e., last allocated element). The previous element is now available through the stack variable followed by an up arrow. If no elements remain in the stack, the stack variable will have the value nil.

<pop statement> ::= pop <stack variable>

⋈ Release 1 does not support stacks.

Examples:

    pop operator_stack

10.4.3 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. 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.

If the execution of a next statement would cause thennew current element to lie outside the bounds of the sequence, then the allocation designator is set tot the value nil.

```
<next statement> ::= next <allocation designator> in <sequence variable>
<sequence variable> ::= <variable>
```

Example:

```
next length_ptr in buf;
next string : [1 .. length_ptr ^] in buf
```

10.4.4 RESET STATEMENT ⋈

The reset statement causes positioning in the sequence. The current element becomes either the first element of the sequence or the element specified by the allocation designator. The use of an allocation designator which was not set by a next statement for the same sequence is an error.

⋈ Release 1 does not support sequences.

<reset statement> ::= reset <sequence variable> [to <allocation designator>]

Example:

reset buf to length_ptr

10.4.5 ALLOCATE STATEMENT

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

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

Examples:

allocate my-stack : [50]
allocate sym_ptr in symbol_table

10.4.6 FREE STATEMENT

The free statement causes the deletion of the specified variable from a heap, thus making its storage available for subsequent allocate statements. If the allocation designator was not set as a result of a previous allocate statement for the same heap the effect is undefined.

* Release 1 supports the universal heap only.

<free statement> ::= <u>free</u> <allocation designator> [<u>in</u> <heap variable>] ⋈

<u>Examples:</u>

<u>free</u> sym_ptr <u>in</u> symbol_table

<u>free</u> my_stack

⋈ Release 1 supports the universal heap only.

10.5 INPUT-OUTPUT STATEMENTS [M]

Two file types are accommodated: binary files which consist of a linear sequence of SWL variables, and text files which consist of a sequence of entities called lines. There is a system defined mapping between lines and string(n) which may differ depending on the source or destination device for the line. Statements that cause transmission from such a file are provided with an additional field to specify the number of characters in the internal representation of the line.

```
<input-output statement> ::= <write binary statement>
                           | <write line statement>
                           | <read binary statement>
                           | <read line statement>
                           | <set mode file statement>
                           | <rewind statement>
                           | <write eof statement>
```

10.5.1 WRITE BINARY STATEMENT

A write binary statement causes the value of an expression to be transmitted to the specified binary file.

<write binary statement> ::= put (<file variable>, <expression>)

Example:

put (intermediate_text, symbol_string)

[M] Release 1 does not support files or the SWL I/O statements.

10.5.2 WRITE LINE STATEMENT

The write line statement causes the transmission of a string value as a line to the specified file.

```
<write line statement> ::= put (<file variable>, <string value>)
<string value> ::= <expression>
```

Examples:

```
put (listing, 'missing loopend symbol')
put (listing, line_buffer)
```

10.5.3 READ BINARY STATEMENT

The read binary statement causes the transmission of a value from the file to a variable. If the sequence of types read is different from the sequence written, the result is undefined. An attempt to read beyond the end of information causes the built in function eof (<file_variable>) to return true.

```
<read binary statement> ::= get (<file variable>, <variable>)
```

Example:

```
get (intermediate_text, symbol_string)
```

N.B. Release 1 does not support files or the SWL I/O statements.

10.5.4 READ LINE STATEMENT

The read line statement causes the next line to be transmitted from the file as a string to the specified variable. The number of characters transmitted is stored in the variable specified by the third parameter. An attempt to read beyond the last line causes the built-in function eof (<file variable>) to return the value true.

<read line statement> ::= get (<file variable>, <string variable>, <no. read>)
<string variable> ::= <variable>
<no. read> ::= <variable>

The string variable must be a variable of type string (n), and no. read must be an integer variable. If the line transmits as more than n characters it is truncated on the right before storing into the string variable.

Example:

get (source_file, line_buffer, line_length)

10.5.5 SET MODE FILE STATEMENT

The set mode file statement sets the mode of the file to read or write mode. A file cannot be written if it is in read mode, and cannot be read in write mode. The file is initially in a neutral mode and is set to read or write mode by the first get or put statement on it.

<set mode file statement> ::= mode (<file variable>, <file mode>)
<file mode> ::= read | write

N.B. Release 1 does not support files or the SWL I/O statements.

Example:

mode (intermediate_text, read)

10.5.6 REWIND STATEMENT

The rewind statement repositions the file at its beginning and sets eof (<file variable>) to false. It has no effect on the file mode.

<rewind statement> ::= rewind (<file variable>)

Example:

rewind (intermediate_text)

10.5.7 WRITE EOF STATEMENT

The write eof statement causes a file in write mode to record information such that on a subsequent attempt to read it beyond the current position the built-in function eof (<file variable>) can return the value true.

<write eof statement> ::= weof (<file variable>)

Example:

weof (intermediate_text)

N.B. Release 1 does not support files or the SWL I/O statements.

11.0 STANDARD PROCEDURES AND FUNCTIONS

Certain standard procedures and functions have been defined for the SWL 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 STANDARD PROCEDURES

The following standard procedures assign values of type string.

~~11.1.1 #translate (s, t, d)~~

~~converts the elements of s according to the translate table t and assigns them to d. S, t and d must all be strings. S and d must have the same length.~~

~~The result is: d(i) = t($int(S(i))) for all elements of d.~~

~~11.1.2 #stringrep (val, substr, width [,decimals])~~

~~converts the value, val, which may be of type integer, real, or boolean to a string representation in "width" characters. The result is stored in the string substr. Integers are represented as decimal numbers with leading zeros blank suppressed. Reals are represented as decimal numbers with exponent base 10, and "decimals" digits after the decimal point. Booleans are represented by true or false, right justified and blank filled.~~

11.2 STANDARD FUNCTIONS

The following standard functions return values of the specified type.

11.2.1 #abs(x)	computes the absolute value of x. The type of x must either be real or integer, and the type of the result is the type of x.
~~11.2.2 #sign(x)~~	~~returns the value 1 if x > 0, the value 0 if x = 0, or the value -1 if x < 0.~~ ~~x must be integer or real, and the result is the same type as x.~~
11.2.3 #succ(x)	x is of any scalar or subrange type, and the result is the successor value of x (if it exists).
11.2.4 #pred(x)	x is of any scalar or subrange type, and the result is the predecessor value of x (if it exists).
11.2.5 $integer(x)	returns the ordinal number of the scalar value x. x must be an ordinal type, char, or real; if x is real then the value returned is an integer y of the same sign as x such that abs(x)-1 < abs(y) <= abs(x).

11.2.6 $real(x)	returns a value of type real that approximates the integer value x. Note that $integer($real(x)) does not necessarily equal x.
11.2.7 $char(x)	x must be an integer value 0 <= x <= 255. The value returned is the character whose ordinal number is x.
11.2.8 $string(l,s[,fill])	returns a string value of length l obtained from the string s by (a) truncating s on the right if length of s > l, or (b) appending characters on the right if length of s < l. The characters appended are blanks, or the character value of fill when it is specified.
11.2.9 #strlength(x)	returns the length in terms of number of characters of the string x. ⋈
11.2.10 #lowerbound (array, n)	returns the value of the n'th lower bound of the array. The type is the index type of that dimension of the array. The left most subscript position is numbered 1. ⋈
11.2.11 #upperbound (array, n)	returns the value of the n'th upper bound of the array. The type is the index type of that dimension of the array. The left most subscript position is numbered 1. ⋈

⋈ Release 1 does not support #strlength, #lowerbound, or #upperbound.

11.2.12 #eof (file) returns the value true if the end-of-file condition exists for the specified file. Returns false otherwise. ⋈

~~11.2.13 #coprecid~~ ~~returns the value of type coproc of the coprocess in which it is executed.~~

~~11.2.14 #rel (pointer [,storage])~~ ~~produces a relative pointer value from a pointer and storage variable. The relative pointer is of the same type as the pointer. The result is undefined if the pointer does not designate an element of the storage variable. If the storage variable is not supplied the default heap is used.~~

~~11.2.15 #ptr (relative_pointer [,storage_variable])~~ ~~is used to convert a relative pointer to a pointer, and is required when using a relative pointer. It returns a pointer of the same type as the relative pointer.~~

~~Example:~~

```
type myheap = heap (50 rep integer);
var  rptr : rel (myheap)  integer,
     h : myheap,
     x : integer;
x := #ptr (rptr, h)^;
```

⋈ Release 1 does not support #eof.

~~11.3 REPRESENTATION DEPENDENT~~

~~11.3.1~~	~~#loc (argument)~~	~~returns a pointer to the argument which can be directly assigned to any pointer type.~~
~~11.3.2~~	~~#size (argument)~~	~~returns the number of cells required to contain a variable of the same type as the argument.~~
~~11.3.3~~	~~#offset(u,base)~~	~~returns an integer value n which is the offset of the variable u in number of cells from an integral multiple of base cell boundary.~~ ~~$0 \leqslant n < \text{base}$.~~
~~11.3.4~~	~~#malignment (argument, offset, base)~~	~~assigns the offset and base alignment required for a variable of the same type as its first argument, to its second and third arguments respectively.~~