LANGUAGE SPECIFICATION

for

SOFTWARE WRITER'S LANGUAGE

Table of Contents

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)

The purpose of this document is to define the Software Writers' Language to such a degree that the language can be understood, used, documented, and implemented by programmers experienced in the use of high-level block-structured languages. The language design and this document are products of the NCR-CDC Software Writers' Language Committee.

The Software Writers' Language will serve as the sole systems programming language for the development of the NCR-CDC Integrated Product Line (IPL) and is established to satisfy the bulk of the IPL systems programming requirements with machine-independent facilities that are well structured and can be implemented efficiently and reliably.

Further language developments can be expected in the areas of operating system, hardware, and debugging facilities as the IPL becomes further defined.

The goals for the Software Writers' Language include:

- It should furnish high functional power for the problems encountered in the creation of compilers and other software systems.
- It should be an "easy to use" language, giving the facility or function needed in a reasonably direct manner.
- As much as possible, it should guard against the programmer's use of programmatic elements in ways which might cause long-undetected errors, or errors if the program were carried from one machine to a different model.
- The language should provide sufficiently high-level constructs to free the programmer from much of the burdensome details of program construction, and sufficient low-level constructs to get the job done.
- SWL should yield effective object programs in computer systems not necessarily yet designed.
- The language should encourage the creation of programs and data whose structure is immediately apparent to the reader.

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)
1.1 INTRODUCTION TO SUBSEQUENT VERSIONS

1.1 INTRODUCTION TO SUBSEQUENT VERSIONS

On acceptance of the first version of the language specification, the original Software Writers Language Committee was dissolved and replaced by a SWL Language Control Board responsible for the future of the language. Concurrently, a SWL Language Log was started for purposes of recording and tracking requests for language changes and their eventual resolution by the board. The SWL specification will be periodically revised to reflect accepted language changes. In addition, minor corrections, rewordings, and clarifications will be made, more examples, exemplary discussion and cross-referencing will be added. This will be a continuing process intended to transform an austere specification, directed to a limited audience, into a more generally usable reference document.

1.1.1 SUMMARY OF CHANGES: REVISION 2 (OCT.'74)

Changes are listed below by topic and section numbers.

- Constants (3.2.4)

An explicit denotation for a base-10 radix has been added.

- Formal Types (4.0 and 4.6)

Procedures, labels, and coprocs are now classified as 'control' types. Formal types include control types, adaptable types, and the newly introduced bound variant record types (cf. 4.7), which are included in the new syntax for SWL types.

- Ordinal types (4.2.1.1.3)

Ordinal type specifications are no longer restricted to appear only in type declarations.

At least two ordinal constant identifiers must appear in the ordinal constant list.

- Subrange Types (4.2.1.1.5)

Variable-bound subranges are no longer permitted.

- Pointer Types (4.2.3)

Read-only pointers--and the associated 'read' attribute for pointers--have been introduced.

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)
1.1.1 SUMMARY OF CHANGES: REVISION 2 (OCT.'74)

Relative pointers may now be used with elements of strings, arrays, and records, as well as with elements of storage types.

The definition of relative-pointer equivalence has been strengthened to include dependence on the 'parental' types of the relative pointers, as well as the types of the elements pointed to.

Specifications for the #rel and #ptr functions have been revised to reflect the new definition of relative pointer equivalence.

- Arrays and Subscripted References (4.3.3 and 6.4.3)

The treatment by SWL of arrays and subscripted references has been clarified; an n-dimensional array may be declared and referenced as a vector of n - 1 dimensional arrays, for example.

- Record Types (4.3.4)

Syntax and semantics for record types have been simplified, and minor corrections made.

Variant records are now legitimate fields of records of fixed type.

Alignment may now be specified for tag fields of variants.

- Union Types (4.3.5)

Union membership is now restricted to distinct, 'valued' types. The non-value types are files and heaps, arrays and stacks of non-value types, and records containing a field of non-value type.

Union types may now be packed.

- Adaptable Types (4.5)

Syntax and semantics for adaptable arrays have been redefined and simplified.

Components of adaptable arrays may now be adaptable types.

Correction to adaptable records now allows field selectors for adaptable fields.

- Procedures, Labels, and Coprocs (4.6 and 8.1)

Formal types are now called 'control' types.

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)
1.1.1 SUMMARY OF CHANGES: REVISION 2 (OCT. '74)

Syntax for procedure type specifications and procedure declaration has been revised; procedure type identifiers may now be used in procedure declarations.

- Bound Variant Record Types (4.7)

Bound variant records may have their case parts 'fixed' to one of their constitutent variants, for purposes of space compaction. They are formal types, which may be used as formal parameters and must otherwise be referenced through pointers.

- Repetition Factors (4.4.2, 10.4.1, 5.1.2)

The spelling for repetitions has been changed to allow parsing without indefinite look-ahead. The definition of spans has been restricted to the use of type identifiers rather than types.

- Type, Constant, and Variable Declarations (4.1, 5.1.1, 6.1)

Empty specifications are now allowed in these declarations. The intent is to provide flexibility in program composition and revision.

- Compilation Units, Modules, Blocks (7)

Syntax and semantics have been reordered and expanded.

Procedures as well as variables are allowed as prongs of compilation units.

Empty declarations are now allowed (for purposes of program composition and revision).

Order of evaluation of declarations (7.9.3) is governed solely by block structuring.

- Expressions (9)

A conditional-and operator (cand) and an unconditional-or operator (uor) have been added.

The pointer type test operator has been respelled (:^:) and an analogous value type test operator (:=:) has been added.

Order of evaluation of expressions (9.3) is now governed solely by the syntactic rules of composition and their implied precedence rules.

- Assignment Statements (10.1)

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)
1.1.1 SUMMARY OF CHANGES: REVISION 2 (OCT.'74)

A successor statement and a predecessor statement have been added.

Stacks and sequences may now be assigned (to stacks and sequences respectively); the restrictions on such assignments are called out (10.1.1, point 8).

Restrictions on assignments involving read-only pointers and pointers to bound variant records are spelled out (10.1.1, points 6, 7).

- Case Statements (10.2.8 and 10.2.9)

Variant case statements have been removed.

A value conformity case statement has been added.

- Formal and Actual Parameters (10.3.1)

Restrictions on the use of bound variant records as actual and formal parameters are spelled out (points 7 and 8).

- Allocation Designators (10.4.1)

Syntax and semantics for allocation designators have been expanded to illuminate the various species of 'fixers' used for adaptables and bound variants.

- Standard Procedures and Functions (11)

The ordering of the parameters of the #translate procedure has been changed.

The following functions now accept a type identifier as well as a variable as an argument: #strlength, #lowerbound, #upperbound, #size, #malignment.

The functions #lowervalue and #uppervalue have been added.

- Compile-Time Facilities (12)

The semantics of the phase-one facilities have been expanded to cover the treatment of identifiers for compile-time variables and macros, and the interpretation of macros and compile-time statements.

- Representation-Dependent Features (13)

The syntax of crammed types has been corrected to preclude the use of variable-bound crammed structures, and to spruce up the

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)
1.1.1 SUMMARY OF CHANGES: REVISION 2 (OCT.'74)

specification of malignment.

1.1.2 SUMMARY OF CHANGES: REVISION 3 (DEC. '74)

Changes are listed below by topic and section numbers.

- Constants (3.2.4)

The $char conversion function is now a <character constant> rather than a <string term>.

- Storage Types (4.4), Storage Mgmt. Statements (10.4)

Mechanisms for accessing elements of a stack other than the topmost one have been added.

The use of a stack identifier followed by an up arrow to reference the top element of a stack is no longer permitted.

A system-defined stack with automatic lifetime characteristics has been introduced.

The push and pop statements have been changed to reflect the above changes.

The reset statement has been extended to permit stacks to be successively popped to a designated element and to permit all elements of a heap to be freed en-masse.

A new built-in function for returning pointers to stack elements has been added (#previous, 11.2.18).

- Pointer Comparisons (cf. 9.2.6.2 and 9.2.6.3)

All six relational operators may now be used with all direct and relative pointers except for pointers to file types, which are non-comparable, and pointers to control types, which may be compared for equality and inequality only.

- Files (4.8, 6.2, 10.5)

File types (4.8) have been rounded out to include print files (formatting controlled by procedures) and direct files (binary files with 'keyed' accessing facilities).

File types are now data types (rather than formal types) which are associated with file variables (6.7). File variables are used to access actual files. An actual file must be associated

with a file variable (by an open statement) in order to be accessed.

'Value constructors' for file variables have been added (5.2.3).

Input-Output statements (10.5) have been extended and revised (in part) to reflect the new file types and the new handling of file accessing. Subsequent revisions will complete these extensions.

1.1.3 SUMMARY OF CHANGES: REVISION 4 (JUNE 1975)

This revision has its changes from Revision 3 specified by 'change bars' in the right margin, except for deletions of entire paragraphs or sections.

Change bars have been manually deleted where they indicate only trivial spelling corrections, clarifications, re-wordings or re-formattings.

Major changes for this revision include:

- The micro facility of the language has been completely removed (but macros remain).

- Input/output has been modified by the addition of formatted output and related functions. See 4.8, 6.7, and 10.5.

- Varying strings and concatenation have been added to the language (cf. 4.3.2, 5.2.1.3, 6.2.9.4, and 11.2.8).

- The use of a procedure-type-identifier in the declaration of a procedure has been removed.

- Restrictions have been placed on type-fixers in allocation designators for variant records (cf. 10.4.1 Allocation Designators).

- A reset command for a storage variable prior to the first allocation into that storage variable is now required (cf. 10.4).

- The spelling of access attributes and file attributes has been changed so they begin with the prefix character '#'.

- Structured conversions have been re-defined (cf. 5.2.1.2).

1.0 INTRODUCTION TO FIRST VERSION (DEC 73)
1.1.3 SUMMARY OF CHANGES: REVISION 4 (JUNE 1975)

- Label declared are required only in ISWL programs, and should not appear in SWL programs (cf. 8.3 Label Declarations).

- An 'alias' for identifiers has been added to the language (cf. 7.7.1 Alias Names).

2.0 LANGUAGE OVERVIEW

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 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

2.0 LANGUAGE OVERVIEW

discipline and notation used to select individual components. Five structuring methods are available: set structure, string structure, array structure, record structure and union structure.

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

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

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

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

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

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.

2.0 LANGUAGE OVERVIEW

Storage types represent structures to which other variables may be added, referenced, and deleted under explicit program control. 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 managed (in a "last in - first out" manner) by the push, pop and reset operations. Stack components are accessed through pointers constructed as by-products of these operations. Sequence types and heap types represent storage structures whose components may be of diverse type. Components of sequences are managed through the operations of resetting to the first component and moving to the next component and are accessed through pointers constructed as by-products of these operations. Space for components of heap storages must be explicitly managed by the operation of allocate and free; the components are accessed through pointers constructed as by-products of the allocate operation.

Many of the structured and storage 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.

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 fixed during execution of the program, through the use of so-called "allocation designators".

An austere set of file types, and 'accessing methods', is provided. Actual files are accessed by means of file variables, and must be explicitly associated with a file variable (through an open statement) in order to be accessed.

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.

2.0 LANGUAGE OVERVIEW

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 to 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.

A procedure is declared in terms of its 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 a 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

2.0 LANGUAGE OVERVIEW

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 specified 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; conformity case statements select one of their component statement lists for execution, depending on the type or the value of a union variable.

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

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.

Input-output statements provide mechanisms for associating (and de-associating) files with file variables (open and close), for positioning files, for reading and writing files, and for explicitly formatting so-called 'print files'.

Finally, assignment statements cause variables to assume new values.

A SWL program is meant to be translated by a compilation

2.0 LANGUAGE OVERVIEW

process into a SWL object program. Object programs resulting from distinct compilations 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: compile-time variable declarations, compile-time statements, and macro facilities.

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 reqdep attribute. The facilities include a cell type, which 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

3.0 METALANGUAGE AND BASIC CONSTRUCTS

3.1 METALANGUAGE

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

Constructs not enclosed in angle brackets stand for themselves.

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

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

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

Examples:

The definition:

```
<real number> ::= <unscaled number>
                | <scaled number>
```

is read, "a real number is defined as either an unscaled number or a scaled number".

The definitions

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

are read, "an unscaled number is a numeral followed by a period followed by a numeral; a numeral is a digit followed by zero or more digits".

The definition

```
<scaled number> ::= <unscaled number>E[<sign>]<numeral>
```

3.0 METALANGUAGE AND BASIC CONSTRUCTS
3.1 METALANGUAGE

is read as, "a scaled number is an unscaled number, followed by the letter 'E', followed by an optional (zero or one) sign, followed by a numeral".

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

Example:

The definition:

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

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

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

Example:

The definition:

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

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

3.2 LEXICAL CONSTRUCTS

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

3.2.1 ALPHABET

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

3.0 METALANGUAGE AND BASIC CONSTRUCTS

3.2.1 ALPHABET

```
<ASCII character> ::= <alphabet>|<unprintable>

   <alphabet> ::= <letter>
                  |<digit>
                  |<special mark>
                  |<blanks>
                  |<unused mark>
                  | ''

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
             |a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

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

<special mark> ::= +|-|*|/|.|;|:|"|'

                   |#|$|_|@|?|(|)|=|<|>

                   |\|[|]|^|!

<blanks> ::=

<unused mark> ::= &|%|{|}|~|`
```

The meaning of '' when occurring within a string term bounded by two single quotes (' and ') is a single quote mark as a part of the string. Outside that context, it has the meaning of a null string (since nothing is bounded). Thus, four single quote (i.e., '''') indicates a string constant with a value of a single quote mark.

3.2.2 IDENTIFIERS

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

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

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

Identifiers are restricted to a maximum of 31 characters, and identifiers that differ only by case shifts of component letters are considered to be identical. Identifiers must begin with a letter and may not contain embedded blanks (cf., Conventions for Blanks below).

3.0 METALANGUAGE AND BASIC CONSTRUCTS
3.2.2 IDENTIFIERS

Examples:

x2 Henry Job# A_wordy_Identifier

Bad_Examples:

1st_character_must_be_a_letter
number_of_characters_must_not_exceed_thirtyone

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, delimiters, 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

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

3.2.5 CONVENTIONS FOR BLANKS

Identifiers, reserved words, and constants must not abut each other, and must not contain embedded blanks. Basic symbols constructed as digraphs or trigraphs 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

4.0 SWL_TYPES

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

```
<SWL type> ::= <data type>
             |<formal type>

<data type> ::= <type>                                  "cf. 4.2"

<formal type> ::= <adaptable type>                      "cf. 4.5"
                |<control type>                         "cf. 4.6"
                |<bound variant record type>            "cf. 4.7"
```

Data types (more briefly, types) are used to define sets of values that can be assumed by SWL variables, their operational domain and--in many cases--a notation for referencing such values (cf. 4.2).

Formal types are used to define objects which must be referenced in an indirect manner; they may be used as formal parameters of procedures and must otherwise be referenced through the use of a pointer mechanism.

Adaptable types and bound variant record types are associated with data types whose precise attributes are meant to be explicitly 'fixed' during execution of the program.

Control types are associated with labels, procedures, and coprocesses.

4.1 TYPE_DECLARATIONS

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

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

```
<type declaration> ::= type [<type spec>{, <type spec>} ]
```

4.0 SWL TYPES

4.1 TYPE DECLARATIONS

<type spec> ::= <type identifier list> = <SWL type> | <empty>

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

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

Type declarations can be used for purposes of brevity, clarity, and accuracy. Once declared, a type may be referred to elsewhere in the program by its declared type identifier which, if properly chosen, can provide a reduction in errors associated with spelling-out type specifications, as well as mnemonic value.

4.2 DATA_TYPES_(TYPES)

Data types (more briefly, types) are used to define sets of values that may be assumed by variables (cf. 6.0).

<data type> ::= <type>

<type> ::= <fixed or variable bound type> | <file type>

Fixed or variable bound types consist of:

a) basic types, which take on simple values;

b) structured types, which define collections of components;

c) storage types, which are used as repositories for collections of components of various types.

File types define actual files of data meant to be manipulated by input-output operations.

4.2.0.1 Fixed or Variable Bound Types

```
<fixed or variable bound type> ::=

  <basic type> ::=
                        <real type>
                      ¦ <pointer type>
                      ¦ <scalar type> ::=
                                          <integer type>
                                        ¦ <character type>
                                        ¦ <ordinal type>
                                        ¦ <boolean type>
                                        ¦ <subrange type>
  ¦ <structured type> ::=

                        <set type>
                      ¦ <union type>
                      ¦ <aggregate type> ::=
                                          <string type>
                                        ¦ <array type>
                                        ¦ <record type>
  ¦ <storage type> ::=

                        <stack type>
                      ¦ <sequence type>
                      ¦ <heap type>
```

A subset of the structured types (the aggregate types) and all the storage types are defined 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 type (cf. 6.1.2.1) 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 (cf. 7.3). For purposes of exposition, the constructs

```
<variable bound type>
```

and

```
<fixed type>
```

are introduced, the latter denoting all types but the former.

4.2.1 BASIC TYPES

Basic types define components that take on simple values, and are the only types that may be associated with 'returned' values of functions (cf. 8.1.3).

```
<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. (cf. 11.2.3, 11.2.4)

```
<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 (cf. 4.2.1.1.5) defined by

$$-n1 \;..\; n2$$

where n1 and n2 denote implementation-dependent integers.

Permissible operations: assignment, set membership test, all six relations, addition, subtraction, multiplication, quotient, remainder, exponentiation, absolute value, built-in-functions (cf. 11).

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 (cf. 4.2.1.1.5) defined by

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

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

Permissible operations: assignment, set membership test, all six relations, built-in function (cf. 11).

4.2.1.1.3 ORDINAL TYPE

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

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

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

An ordinal type defines an ordered set of values by enumeration, in the ordinal list, of the identifiers which denote the values. Each of the identifiers (at least two) in the ordinal list is thereby declared as a constant of the particular ordinal type.

Permissible operations: assignment, set membership test, all six relations, built-in functions (cf. 11).

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

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

4.0 SWL TYPES
4.2.1.1.3 ORDINAL TYPE

relations hold:

red < green
red < blue
green < blue

A mapping from ordinals onto non-negative integers is provided by the $integer function (cf. 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.

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 conceptually equivalent to the ordinal type specified by the ordinal list

(false,true)

Permissible operations: assignment, set membership test, all six relations (false < true), sum, product, difference, symmetric difference, negation, built-in functions (cf. 11).

4.2.1.1.5 SUBRANGE TYPE

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

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

4.0 SWL TYPES
4.2.1.1.5 SUBRANGE TYPE

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

Equivalence rules are relaxed for subranges to permit values from a subrange and values from its parent range (or another subrange of its parent range) to be assigned to each other and to enter into the operations of assignment and comparison, and other dyadic operations.

Permissible_operations: as for the parent type.

Example:

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

The ordinal 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 (cf. Standard Functions, 11.2).

Permissible_operation: assignment, all six relations, addition, subtraction, multiplication, quotient, built-in functions (cf.11).

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 types represent locations of instances of objects of SWL type.

Relative pointer types represent locations of components of objects of storage type or aggregate type relative to the variable of storage or aggregate type.

Permissible operations: assignment, union membership (cf. 9.2.1), all six relations except for pointers to file (non-comparable) and pointers to control types (comparable for equality and inequality only), built-in functions (cf.11).

4.2.3.1 Direct Pointer Types

Direct pointer types are introduced by an up arrow, followed by a SWL type to which the pointers are always bound; direct pointer variables may assume, as values, only pointers to that SWL type.

```
<direct pointer type> ::= <pointer to type>
                         |<formal pointer>

<pointer to type> ::= ^<fixed or variable bound type>
                     |<pointer to file>

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

<formal pointer> ::= <adaptable pointer>
                    |<pointer to control>
                    |<bound variant pointer>

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

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

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

Formal pointers provide the sole mechanism for accessing objects of formal type, other than through formal parameters of

procedures. In particular, adaptable pointers and bound variant pointers are used to access instances of adaptable variables and bound variant records whose type has been 'fixed' by an allocate or a next statement (cf. Sections 10.4, 10.4.3, 10.4.5).

See Section 10.1, Assignment Statements, for rules governing pointer assignment.

Direct pointers are equivalent if they are defined in terms of equivalent types. No pointer to type can ever be equivalent to a formal pointer.

The following ancillary constructs are introduced for expository use elsewhere in the document.

```
<pointer to procedure> ::= <formal pointer>
<pointer to coproc> ::= <formal pointer>
<pointer to label> ::= <formal pointer>

<adaptable pointer to string> ::= <adaptable pointer>
<adaptable pointer to array> ::= <adaptable pointer>
<adaptable pointer to record> ::= <adaptable pointer>
<adaptable pointer to stack> ::= <adaptable pointer>
<adaptable pointer to sequence> ::= <adaptable pointer>
<adaptable pointer to heap> ::= <adaptable pointer>
```

4.2.3.2 Relative_Pointer_Types

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

```
<relative pointer type> ::=
     rel[(<parental type>)]^<object type>

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

<aggregate type> ::= <string type>|<array type>|<record type>

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

Relative pointer types are equivalent if they are defined in terms of equivalent parental types and equivalent object types. If the parental type is not specified, a default, system-defined heap is assumed.

Relative pointers provide three facilities not given by direct pointer types:

4.0 SWL TYPES
4.2.3.2 Relative Pointer Types

1. A relative pointer variable requires less space than a direct pointer variable.

2. A linked list or array of relative pointers (or a similar pointer network) related to a parental variable is still correct if that entire variable is assigned to another variable of the same parental type.

3. A relative pointer variable (or group of relative pointer variables) may be used to refer simultaneously to several variables of the same parental type where parallel information is contained (so long as the parental type is not a heap).

Relative pointer values can be generated solely through the built-in function #rel (cf. 11.2.14) whose arguments are a pointer variable and an optional parental variable.

Relative pointers cannot be used to access data directly. Such data must be accessed through a direct pointer generated by the built-in function #ptr (cf. 11.2.15) whose arguments are a relative pointer variable and an optional parental variable.

Example:

```
type intrel = rel (heep#tipe) ^rec#tipe,
rec#tipe = record
             fa : integer,
             fb : real,
             fc : string (20) of char,
             fwd : intrel, "relative pointers to records"
             bkwd : integer "within stacks of heep#tipe"
           recend ;

type heep#tipe = heap (rep 100 of rec#tipe).
type latest = stack [10] of intrel ;

"latest is a small stack of relative pointers which
can point to records within heaps of heep#tipe"
```

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

structure. Each will be described in the sequel.

```
<structured type> ::= <set type>
                    |<union type>
                    |<aggregate type>

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

Aggregate types may be of variable bound type (cf. 4.2).

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

Permissible operations: assignment, intersection, union, difference, symmetric difference, negation, inclusion, identity.

Set types are equivalent if they have equivalent base types.

Example: The set, akcess, declared by

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

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

```
$akcess [] "the empty set"
$akcess [no_read]
$akcess [no_write]
$akcess [no_execute]
$akcess [no_read, no_write]
$akcess [no_read, no_execute]
$akcess [no_write, no_execute]
$akcess [no_read, no_write, no_execute]
```

where the notation "$akcess [...]" denotes a value constructor (cf. Value Constructors, Section 5.1) for the set type, akcess.

4.3.2 STRING TYPE

A string type represents ordered n-tuples of values of character type. Two string types are provided: fixed strings and varying strings.

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

<fixed string> ::= string (<length>) of <character type>

<varying string> ::=
     vstring (<maxlength>) of <character type>

<length> ::= <positive integer expression>

<maxlength> ::= <positive integer expression>

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

A fixed string of length n represents all ordered n-tuples of values of character type. A varying string whose maxlength is n represents all fixed strings of length k (1 <= k <= n) together with the null_string (see below).

The current_length of a string is defined as follows: The current length of a varying string is defined to be m whenever its value is a fixed string of length m and is defined to be zero whenever its value is the null string. The current length of a fixed string is equal to its length. The function #curstrlength (cf. 11.2.21) returns the current length of a string or a varying string. The function #strlength (cf. 11.2.9) returns the maxlength of a varying string and the length of a fixed string.

A constant denotation for the null string (whose maxlength or length is zero) is provided:

```
<null string> ::= ''
```

An ordered k-tuple of the values of a string or varying string (1 <= k <= n) is called a substring. Notation for accessing substrings is provided (cf. 6.4.2, Substring References).

4.0 SWL TYPES
4.3.2 STRING TYPE

Two string types are equivalent if they are fixed strings of the same length, or are varying strings of the same maxlength. In the case of a variable length (or variable maxlength), the length (or maxlength) is determined when the declaration is elaborated (cf. 4.2 Fixed and Variable-Bound Types).

Permissible operations: assignment, comparison (all six relational operators), concatenation, and the built-in functions (cf. ch.11). Equivalence rules are relaxed to permit fixed strings and varying strings to enter jointly into these operations, independent of their current lengths, with truncation or right extension with blanks carried out when necessary. In addition, characters are treated as strings of length one (1) when they enter into these (but only these) operations (see also assignments in 10.1, 10.1.1, 10.1.3, and string-values as factors in 9.1).

Warning: Despite the special exceptions of the preceeding paragraph, the types string, vstring, and char are not equivalent. In particular, a pointer to one of those types may not be assigned to a pointer to another of the types, or enter into other operations with pointers to those other types; nor may an actual parameter of one of the types be passed to a procedure where the corresponding formal parameter is a reference parameter of another type.

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 n index values whose types are indicated by the indices in the definition.

```
<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>
           |<scalar expression> .. <scalar expression>

<component type> ::= <type>

<packing> ::= <packing attributes>
```

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

4.3.3.1 Array Dimensionality and Equivalence

If the component type is not an array type and n indices are specified, then the array type has dimension n. If the component type is an array type of dimension m, and n indices are specified, then the array has dimension n + m. 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.

4.3.3.2 Alternate Spellings for Array Types

There are 2 ** (n - 1) distinct spellings for specifying an array of dimension n. For example, for n = 3:

```
array [int] of array [int,int] of char
array [int,int] of array [int] of char
array [int,int,int] of char
```

are all equivalent spellings for:

```
array [int] of array [int] of array [int] of char
```

which is the spelling that precisely defines SWL's treatment of arrays.

Similar alternative spellings are allowed for referencing array components (cf. Subscripted References, 6.6.3).

4.3.3.3 Packed Arrays

Packing attributes (cf. 4.9) are used to specify storage space -- access time tradeoffs for array components. Components of a packed array will be mapped onto storage so as to conserve storage space at the expense of access time. The array itself (the collection of components) is always mapped onto an addressable memory location (i.e., the array is aligned) unless the array itself is an unaligned element of a packed structure.

Example:

```
var i,j : integer;
```

4.0 SWL TYPES

4.3.3.3 Packed Arrays

```
type  hotness = array [color] of non_negative_integer,
      token_code = array [enar] 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,
      array2b = array [i1] of array [i2] of s1,
      array3 = array [i..j] of boolean,
      array4 = array [1..10] of array3;
```

The array types 'array1,' 'array2,' and 'array2b' are equivalent. The 'array3' type is of variable bounds (because its index range cannot be determined until run-time elaboration of the declaration). This holds in a similar way for the 'array4' type, since its component type is 'array3.'

4.3.4 RECORD TYPE

A record type represents a structure consisting of a fixed number of components called fields. Fields are defined in terms of their types and associated field selectors, which are identifiers uniquely denoting that field among all other fields of the record (cf. 6.4.4, Field References).

Permissible operations: assignment, comparison for equality and inequality only; however, variant records (see below) can not be compared.

Records are classified as being either fixed records or variable bound records.

```
<record type> ::= <fixed record type>
                 |<variable bound record type>
```

4.3.4.1 Fixed_Records

Fixed records, which include both invariant records and variant records, will always be allocated a fixed amount of space.

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

4.3.4.2 Invariant_Records_and_Fixed_Fields

An invariant record contains only fixed fields, which are fields of fixed type (cf. 4.2).

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

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

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

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

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

4.3.4.3 _Variable_Bound_Records_and_Variable_Bound_Fields

A variable bound record consists of zero or more fixed fields followed by one and only one variable bound field, which is a field of variable bound type (cf. 4.2).

```
<variable bound record type> ::=
      [<packing>]<variable bound record type identifier>
     ![<packing>]<variable bound record spec>

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

<variable bound field> ::=
      <field selector> : [<alignment>]<variable bound type>

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

4.3.4.4 Variant Records and Case Parts

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

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

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

<recend> ::= [,]recend

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

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

<variations> ::= <variation> {, <variation>}
<variation> ::= =<selection values>= <variant>

<selection values> ::= <selection value> {, <selection value>}
<selection value> ::=
  <constant scalar expression>[..<constant scalar expression>]

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

4.3.4.5 Record Type Equivalence

Two record types are equivalent if they have the same packing (cf. 4.9), the same number of fields, identical field selectors, and equivalent types for corresponding fields. Two variants are equivalent 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.

4.3.4.6 Adaptable_and_Bound_Variant_Record_Types

Two further types of records are adaptable record type and bound variant record type (cf. 4.5.3 and 4.6.4). These are formal types (cf. 4.0) which can be used as formal parameters of procedures but must otherwise be referenced through pointer mechanisms.

4.3.4.7 Packed_Records,_Aligned_Fields

Packing and alignment attributes (cf. 4.9) are used to specify storage space -- access time tradeoffs for fields of records. Fields of packed records are mapped onto storage so as to conserve space at the expense of time. However, aligned fields are mapped onto storage so as to be directly addressable. Records themselves (the collection of fields) are always aligned, unless they are unaligned fields of a packed structure.

Example:

```
type
      date = record day : 1..31, "date is an"
                                 "invariant record type"
                  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,
          "redbook might be a variable bound record type"
                       status : status,
                       scores : array[0..n] of date
                 recend,

      shape = (triangle, rectangle, circle),
      angle = -180..180,
      figure = record x, y, area : real, "figure is a variant"
                                         "record type"
```

```
case s : shape of
= triangle = side : real, inclination, angle1,
  angle2 : angle
= rectangle = side1, side2: real, skew, angle3:
  angle
= circle = diameter: real
casend
recend;
```

4.3.5 UNION TYPE

Union types represent a finite set of selectable, non-equivalent types, and are used to define variables that can take on values of different types.

Such values-of-the-moment can not be accessed in a direct manner. However, the type-of-the-moment of a union variable can be compared with the type of a variable, and its value-of-the-moment (or, optionally, a pointer to it) assigned to that variable, if the types match (cf. Type Testing Operators and Conformity Case Statements), so that a program branch appropriate to values of that type can be executed.

```
<union type> ::= [<packing>] union (<members>)
<members> ::= <type list>
<type list> ::= <type> {, <type>}
```

Permissible operations: assignment, union membership (cf. Type Testing Operators, 9.2.1). In addition, union variables may be used in conjunction with Conformity Case statements (cf. 10.2.8, 10.2.9) for selective execution of statements dependant on the type of value last assigned to the union variable.

4.3.5.1 Restrictions on Union Membership

The so-called non value types cannot be members of a union. The non-value types are heaps, arrays and stacks of non-value type components, and records containing a field of non-value type.

If vt1 and vt2 are variables of the types t1 and t2 respectively, and if the assignment statement vt1 := vt2 is potentially admissible within the scope of a union type declaration, then t1 and t2 cannot both be members of that union type.

4.0 SWL TYPES
4.3.5.1 Restrictions on Union Membership

Example:

```
        " an improper union "
    begin
    var n, r, s, t : integer ;
        t := ... ; ... ;
        n := k - t ; r := n + s ;

        begin
            type mixture =
                  union (boolean, real, vecta, vectb, ord) ;
            type ord = (kread, kwrite, kclose, kopen),
                 vecta = array [1..99] of integer,
                 vectb = array [n..r] of integer ;

    "either vecta or vectb must be removed from the
     union type mixture, since potential equality of type
     could occur, depending on the values of n and r."

    ... ; end end
```

4.3.5.2 Packed Unions

A union type is in reality a type of two components--the type field and the value field. The packing (cf. 4.9) of a union variable affects the alignment of these fields but does not affect the packing of the union members. A union type that is packed because it is declared to be packed or because it is a member of a packed structure such as an array or a record cannot be made the object of the pointer type test operator (cf. 9.2.1) nor used in a pointer conformity case statement (cf. 10.2.9.1) because, by definition, the fields of such a union are not addressable.

4.3.5.3 Union Type Equivalence

Two union types are equivalent if and only if they have the same packing attribute and the two ordered sequences of types are pair-wise equivalent.

4.4 STORAGE TYPES

Storage types represent structures to which other variables may be added, deleted, and referenced under explicit program control (cf. Storage Management Statements, 10.4).

<storage type> ::= <stack type>

4.0 SWL TYPES
4.3.5.1 Restrictions on Union Membership

Example:

```
        " an improper union "
    begin
    var n, r, s, t : integer ;
        t := ...  ; ...  ;
        n := k - t ; r := n + s ;

        begin
            type mixture =
                  union (boolean, real, vecta, vectb, ord) ;
            type ord = (kread, kwrite, kclose, kopen),
                 vecta = array [1..99] of integer,
                 vectb = array [n..r] of integer ;

    "either vecta or vectb must be removed from the
     union type mixture, since potential equality of type
     could occur, depending on the values of n and r."

    ... ; end end
```

4.3.5.2 Packed Unions

A union type is in reality a type of two components--the type field and the value field. The packing (cf. 4.9) of a union variable affects the alignment of these fields but does not affect the packing of the union members. A union type that is packed because it is declared to be packed or because it is a member of a packed structure such as an array or a record cannot be made the object of the pointer type test operator (cf. 9.2.1) nor used in a pointer conformity case statement (cf. 10.2.9.1) because, by definition, the fields of such a union are not addressable.

4.3.5.3 Union Type Equivalence

Two union types are equivalent if and only if they have the same packing attribute and the two ordered sequences of types are pair-wise equivalent.

4.4 STORAGE TYPES

Storage types represent structures to which other variables may be added, deleted, and referenced under explicit program control (cf. Storage Management Statements, 10.4).

<storage type> ::= <stack type>

| <sequence type>
| <heap type>

Storage types may be of variable bound type (cf. Fixed and Variable Bound Types).

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 managed (using a 'last in-first out' discipline) by the push, pop, and reset operations (cf.10.4.2, 10.4.3, and 10.4.7). In addition, stacks may be assigned to stacks; no other operations are allowed.

Two stacks are equivalent if they have the same size and component type.

Stack components are accessed through pointers constructed as by-products of the push and pop operations. The 'Nth' element below the top (current) element of the stack may be accessed by use of the standard function #previous (cf. 11.2.18).

A system-defined stack (cf. 10.4.2.2) is provided. Components of any type may be allocated (pushed) on this stack, but can not be explicitly freed (popped or reset) by the user. Instead, such freeing is done automatically on exits from blocks (cf. 7.5).

4.4.2 SEQUENCE TYPE

<sequence type> ::= seq (<space>) "cf. 4.4.4"

A sequence type represents a storage structure whose components are referenced (by a sequential accessing discipline) through pointers constructed as by-products of the next and reset operations (cf. 10.4.4, 10.4.5). In addition, sequences may be assigned to sequences; no other operations are allowed.

4.4.3 HEAP TYPE

<heap type> ::= heap (<space>) "cf. 4.4.4"

A heap type represents a structure whose components can be explicitly allocated (by the allocate statement) and freed (by the free and reset statements), and which are referenced by pointers constructed as by-products of the allocate statement (cf. 10.4.5, 10.4.6). No other operations on heaps are allowed.

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

4.4.4 SEQUENCE AND HEAP SPACE

<space> ::= {,}

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

A space attribute of the general form

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

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

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

4.5 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 (cf. Procedure Type, 4.6.2) and adaptable pointers (cf. Pointer Type, 4.2.3), the latter providing a mechanism for referencing variables of such types.

4.0 SWL TYPES
4.5 ADAPTABLE TYPES

Adaptable types represent classes of related types to which they can adapt. Adaptation to such an instantaneous type (cf. 6.1.2.1) can occur in three distinct ways:

Adaptable types can be explicitly fixed by the use of allocation designators associated with storage management statements (cf. 10.4).

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

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

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

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

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

4.5.1 ADAPTABLE STRING

Adaptable fixed strings (varying strings) can adapt to fixed strings (varying strings) of any length (maxlength).

```
<adaptable string> ::= <adaptable fixed string>
                     | <adaptable varying string>
                     | <adaptable string identifier>

<adaptable fixed string ::= string (*) of <character type>

<adaptable varying string> ::= vstring (*) of <character type>

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

Two adaptable fixed strings are always equivalent, and two adaptable varying string are always equivalent.

4.5.2 ADAPTABLE ARRAY

Adaptable arrays may have either adaptable components or one (or more) adaptable indices, or both.

Adaptable arrays can adapt to any array with the same packing attribute, the same types of subscripts and either the same component type or (in the case of an adaptable component type) a type to which the adaptable component type can adapt.

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

<adaptable array identifier> ::= <identifier>

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

<starred list> ::=
      {<index>,} <starred index> {, <any index>}

<any index> ::= <index> | <starred index>
<starred index> ::= <star> | <starred subrange>
<star> ::= * | * : <scalar type>
<starred subrange> ::= * .. <scalar expression>
                     |<scalar expression> .. *

<adaptable component type> ::= <adaptable type>
```

Note that component types, indices, and the upper and lower expressions associated with starred subranges may be of variable bound type.

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

Adaptable arrays are equivalent if they have the same packing (cf. 4.9), equivalent component types, identical dimensions, and if corresponding indices and elements of starred lists are equivalent. Two stars are equivalent if they have the same associated types; two starred subranges are equivalent if their lower and upper expressions are equivalent.

4.5.3 ADAPTABLE RECORD

Adaptable records consist of zero or more fixed fields (cf. 4.3.4) followed by one and only one adaptable field, which is a field of adaptable type.

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

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

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

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

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

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

Two adaptable record types are equivalent if they have the same packing (cf. 4.9), the same number of fields, and identical field selectors and equivalent types for corresponding fields.

4.5.3.1 Bound_Variant_Record

A bound variant record is a variant record whose case part is meant to be fixed to one of its constituent variants. See Section 4.7 for syntax and semantics.

4.5.4 ADAPTABLE STACK

Adaptable stacks can adapt to a stack of any size, with the same component type.

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

<adaptable stack identifier> ::= <identifier>
```

The maximum number of components of an adaptable stack can be fixed by a length fixer (cf. 10.4).

4.5.5 ADAPTABLE SEQUENCE

Adaptables sequences can adapt to a sequence of any size.

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

<adaptable sequence identifier> ::= <identifier>

The space for an adaptable sequence can be fixed by a span fixer (cf. 10.4).

4.5.6 ADAPTABLE HEAP

Adaptable heaps can adapt to a heap of any size.

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

<adaptable heap identifier> ::= <identifier>

The space for an adaptable heap can be fixed by a span fixer (cf. 10.4).

4.6 CONTROL_TYPES

<control type> ::= <label type>
| <procedure type>
| <coprocess type>

Control types may be used solely to define reference parameters (cf. below) and pointers (cf. Pointer Type, 4.2.3).

4.6.1 LABEL TYPE

Labels are identifiers used for referring to specific statements (cf. 10.0). Refer to sections 8.3 and 10.0 for the semantics of labels.

<label type> ::= label

4.6.2 PROCEDURE TYPE

Procedures are identified portions of programs that can be activated on demand. Refer to section 8 and 10.3 for the semantics of procedures.

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

```
<procedure type> ::= <procedure type identifier>
                   |proc <proc type spec>

<proc type spec> ::=
    [[<proc type attributes>]][<parameter list>][<return type>]

<proc type attributes> ::=
      <null construct (for expansion purposes)>

<procedure type identifier> ::= <identifier>

<parameter list> ::= (<param segment>{;<param segment>})
<param segment> ::= <reference params>|<value params>

<reference params> ::=
     ref <reference param> { ,<reference param> }

<reference param> ::=
      <formal param list> : [[ #read ]] <ref type>

<value params> ::= val <value param>{ ,<value param>}

<value param> ::=
      <formal param list> : [[ #read ]] <val type>

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

<ref type> ::= <SWL type>
<val type> ::=
     <type> | <adaptable type> | <bound variant record type>

<return type> ::= <basic type>
```

Val type is further restricted to exclude the so-called non-value types: heap types, arrays and stacks 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 (ref or val), and equivalent types, and if their return

types are equivalent. The #read access attribute (cf. 7.1.1.1) defines a read-only parameter.

The presence of a return type in the proc type spec indicates that the procedure is a functional procedure to be invoked as a factor, rather than by a procedure call statement (cf. 8.1.3, Functions, and 9.0, Expressions).

4.6.3 COPROCESS TYPE

Coprocesses are procedures controlled as synchronous processes, so that partial execution of more than one procedure, with a single thread of control passing back and forth between them, is permitted. Refer to sections 8.2 and 10.3 for semantics of coprocs.

<coprocess type> ::= coproc

4.7 _BOUND_VARIANT_RECORD_TYPE

A bound variant record is a variant record (cf. 4.3.4) whose case part is meant to be fixed to one of its constituent variants by the use of a tag field fixer (cf. Section 10.4). These are space saving constructs, since only the space required for the selected variant is allocated.

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

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

<recend> ::= [,]recend

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

4.0 SWL TYPES
4.7 BOUND VARIANT RECORD TYPE

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

<variations> ::= <variation> {, <variation>}
<variation> ::= =<selection values>= <variant>

<selection values> ::=
      <selection value> {, <selection value>}
<selection value> ::= <constant scalar expression>
                        [ .. <constant scalar expression>]

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

A bound variant record type may only be used to define formal parameters or to define pointers for bound variant record types (i.e., bound variant pointers). Thus a variable of this type is always either a formal parameter of a procedure, or is allocated (cf. 10.4) in a sequence or a heap, or in the system-managed stack.

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

If the formal parameter of a procedure is of bound variant record type, then the actual parameter may be of either variant record type or bound variant record type.

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

A bound variant record type is never equivalent to a variant record type.

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

4.8 FILE_TYPES

Files are sources and sinks of data that can be accessed through input-output statments. Four file types are provided.

4.0 SWL TYPES
4.8 FILE TYPES

<file type> ::= legible | print | binary | direct

Legible files consist of a sequence of entities called lines. System-defined mappings between lines and strings_of_char exist; these may differ depending on the source or destination device of the lines.

Print_files are special cases of legible files that permit the user to control output formatting through the use of pagination, spacing and titling procedures (permitted on print files only), rather than through the use of embedded control characters. The user should not directly embed such control characters in data destined for print files.

Binary_files consist of a linear sequence of SWL variables. These variables are not self-identifying, so that results of a read operation are guaranteed if and only if the sequence of types read is the same as the sequence written.

Direct__files are special cases of binary files that also permit the retrieval (and rewriting) of variables 'directly' through the use of a 'key'. Results of such a read (or rewrite) operation are guaranteed if and only if the obvious (but tediously described) type matching holds.

Files are referenced by so-called input output statements (cf. 10.5) which refer to files via file-variables.

4.8.1 FILE VARIABLES

Files are accessed through file__variables, which are associated with a file by an explicit open procedure and de-associated from a file by an explicit close procedure. File variables take on as values some undefined structured collection of values which defines the instantaneous state of the actual file associated with the variable. They may be used as components of arrays and fields of records, may be allocated in storage variables, and may be assigned to other file variables of the same type. In addition, they may be used as actual parameters of procedures, and file types may be used to define both ref and val formal parameters. In general, then, file variables are data variables with a restricted operational domain. In particular, file variables can not be compared.

4.8.2 FILE VARIABLE WARNING

Warning: If a file variable is assigned to another file variable, and either used for manipulation of a file, then resulting changes in the file variable used are not implicitely reflected in the other file variable. The use of both file variables for simultaneous manipulation of the same file could result in serious errors.

4.9 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.

Elements of packed structures are not guaranteed to lie on addressable memory sites (i.e., pointers to such elements can not always be generated). The aligned attribute must be used to ensure addressability of such elements. Addressability is achieved at the expense of storage space (except in certain serendipitous cases), so that the effect of packing may be diluted, sometimes severely.

Explicitly unpacked structures and their components are always aligned. Packed structures are also aligned unless they are unaligned components of a packed structure, but their components are not unless they are explicitly given the aligned or unpacked attribute.

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

4.10 OTHER ASPECTS OF TYPES

4.10.1 INSTANTANEOUS TYPES

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

4.10.2 VALUE AND NON-VALUE TYPES

Value assignments (cf. Assignment Statements) are permitted only to variables of the so-called value types. The non-value types are:

a) heaps;
b) arrays of non-value component types
c) stacks of non-value component types
d) records containing a field of non-value type.

4.10.3 COMPARABLE AND NON-COMPARABLE TYPES

Value comparisons (cf. Relational Operators) are permitted only between variables of the so-called comparable types. The non-comparable types are:

a) files, stacks, heaps, sequences, unions and variant records;
b) arrays of non-comparable component types;
c) records containing a field of non-comparable type.

4.10.4 FUNCTION-RETURN TYPES

The only types that can be associated with returned values of functions (cf. Functions and Return Types) are the basic types:

a) integer, char, boolean, ordinal types, sub-range types;
b) real types;
c) pointer types.

4.10.5 CONVERTIBLE AND CONFORMABLE TYPES

Mechanisms for converting values of some types to values of others are provided (cf. Value Conversion).

a) Scalar values and real values are convertible to integer values, and conversely;
b) Conversions are allowed between conformable arrays and between conformable records (cf. Conformable Arrays and Records).

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS

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 and strings. 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

5.1.1.1 Constants

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

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

<basic constant> ::= <scalar constant>
                   |<compile time variable> "cf. Section 12.1"
                   |<real constant>
                   |<pointer constant>

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

<ordinal constant> ::= <ordinal constant identifier>
     "cf. 4.2.1.1.3"

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

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

<character constant> ::= '<alphabet>'
```

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS
5.1.1.1 Constants

```
                        |<character constant identifier>
                        |$char (<integer>)
                         "cf. Standard Functions, 11.2"

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

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

<string term> ::= <character constant>
                 |<string constant identifier>
                 |'<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 | 10 | 16

<sign> ::= + | -
```

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

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS
5.1.1.2 Constant Expressions

5.1.1.2 Constant Expressions

Constant expressions are constructs denoting rules of computation for obtaining values (at compile time) by the application of operators to operands. The rules of application are those for expressions (cf. 9.0) with the following constraints:

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

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

c) The only functions allowed as factors in such expressions are the $integer, $char, $boolean, #abs, #sign, #succ, #pred, and $<scalar type identifier> functions with constant expressions as arguments.

d) Real constant expressions used in constant declarations are constrained to be either a <real number> or a <real constant identifier>.

5.1.1.3 Constant Declarations

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

```
<constant declaration> ::=
      const [<constant spec> {, <constant spec>}]

<constant spec> ::=
      <constant identifier list> = <constant expression>
     | <empty>

<constant identifier list> ::= <identifier list>
```

A constant spec associates one or more identifiers with the value of the constant expression.

5.1.2 DEFINITE VALUE CONSTRUCTORS

Definite value constructors are used to denote instances of values of a specified set, array, union, or record type, and to

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS
5.1.2 DEFINITE VALUE CONSTRUCTORS

denote instances of typed empty sets and typed 'nil' pointers.

```
<definite value constructor> ::=
     $<constructor id> [<value elements>]
   ! $<set type identifier> [ ]    "the empty set"
   ! $<pointer type identifier> [ nil ]
   ! $<union type identifier> [ <expression> ]

<constructor 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>] *

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

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 target type, except for 'undefined elements,' which are denoted by an asterisk, '*'. Definite value constructors can be used wherever an expression can be used, with the caveat that 'undefined fields' may yield results which are either undefined or erroneous, or both.

Initialization of, or assignment to, a union variable requires that the right hand side's type be known. When that type is to be a pointer type with value nil, the $<pointer type identifier> for nil as shown in syntax above may be used.

The expression used in a union value constructor must evaluate to a value whose type can be assigned to one (and only one) member of that union.

Rep specs may be used solely for array construction.

Note that a set value may be defined to be 'empty' by use of nothing between the brackets [and].

All fields of a definite value constructor corresponding to tag fields of a variant record must be constant scalar expressions.

5.1.3 INDEFINITE VALUE CONSTRUCTORS

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

```
<indefinite value constructor> ::= [<value elements>]
                                 | []   "the empty set"
```

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

All fields of an indefinite value constructor corresponding to tag fields of a variant record must be constant scalar expressions.

The lack of value elements can be used to define the indefinite value of 'an empty set', when nothing appears between the square brackets [and].

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[[rep 3 of true], 'SBC']
   -------+-------  --+--
          |           +-----<string constant> for field s
          +-----<indefinite value constructor> for field t
```

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS
5.1.3 INDEFINITE VALUE CONSTRUCTORS

```
$R2[[red, blue],'CBS',[rep 20 of 2],[[rep 3 of false],'BCS']]
    -----+------ --+-- -----+--------  --------+---------------
         |         |        |                  +--<indefinite value
         |         |        |                      constructor> for
         |         |        |                      F4
         |         |        +--<indefinite value constructor>
         |         |            for F3
         |         +--<string constant> for F2
         +--<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.

Example:

```
type vio = record
             inout : (kread, kwrite, kopen, kclose) "ordinal
                                                     type" ,
             index : integer,
             fname : string (8) of char
           regend ;
           var arec : vio ;
             ...
           arec := $vio[kopen, * , 'holdings'] ;
           "value of field arec.index is undefined"
```

5.2 VALUE_CONVERSION

Most dyadic operations are defined only for operands of the same type. This requirement is relaxed to permit:

a) values of different subranges of the same parent type (and values of the parent type) to enter jointly into operations defined for the parent type;

b) characters to be used with strings in string operations;

c) other formally non-equivalent, but operationally similar types to enter jointly in some cases of assignment (cf. 10.1.1) and 10.3.1) and comparison (cf. 9.2.6).

In all other cases, when it is necessary to operate on dis-similar types the conversion functions described below must be used.

5.2.1 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 convertible to the target type.

5.2.1.1 Basic Conversions

These consist of the 'pre-defined' functions (cf. Standard Functions, 11.2).

```
$integer (<real expression or char expression or ordinal
   expression or boolean expression>)
$real (<integer expression>)
$char (<integer expression>)
$boolean(<integer expression>)
```

and the 'definable' functions:

```
$<ordinal type identifier>(<integer expression>)

$<integer type identifier>(<real expression or char
          expression or ordinal expression
          or boolean expression>)

$<real type identifer(<integer expression>)
$<char type identifier> (<integer expression>)
$<boolean type identifier>(<integer expression>)
```

Conversions between the basic types are the conventional ones and are defined in Section 11.2.

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

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS

5.2.1.1 Basic Conversions

Examples:

```
type status = (nowclose, nowopen, nowread);
var i2, i3, i4, i5 : integer,
    r2, r3 : real, b, babbage : boolean,
    stat : status,
    ch1, ch2 : char ;

    i2 := 2; i3 := 3; i4 := 4;
    r2 := 2.2; r3 := -3.3 ; stat := nowread;
    i5 := 1; babbage := false;
    i2 := $integer( r3 ); "new value of i2 is -3 "
    i1 := $integer(stat); "i1 now = 2."
    ch1 := $char(i2 ); "ch1 now = 2nd ASCII character "
    stat := $status(1) ; "stat value is changed to nowopen "
    i4 := $integer( babbage ); "i4 set to zero "
    b := $boolean(i5); "b set to true"
```

5.2.1.2 Conformable Array and Record Conversions

Array-to-array and record-to-record conversions are defined only for arguments that are 'conformable' to the target type.

$<array type identifier>(<array expression>)

$<record type identifier>(<record expression>)

```
<array expression> ::= <expression>
                     !<indefinite value constructor>

<record expression> ::= <expression>
                      !<indefinite value constructor>
```

Conformability is defined recursively, in terms of the conformability of array components and fields of records, by the following table.

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS
5.2.1.2 Conformable Array and Record Conversions

Target Component or Field Type	Conformable Component or Field Type
basic types	any type assignable to target
set types	any type assignable to target
string types	any type assignable to target
union types	any type assignable to target
array types	array type whose indices span same number of elements as target type and whose component type conforms to target component
array types	indefinite value constructor containing same number of elements as index type of target, all conformable to target component.
non-variant record types	non-variant record type or indefinite value constructor: same number of fields, with each field conformable to corresponding target field.
variant record types	equivalent record type; or indefinite value constructor with same number of fields, each field conformable to the corresponding target field, with constant scalar expressions for tag fields.

5.0 VALUE CONSTRUCTORS AND VALUE CONVERSIONS
5.2.1.2 Conformable Array and Record Conversions

Example:

```
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 : integer,
                beta : real,
                gamma : A2
           recend,

      R3 = packed R2;
```

The two array types conform and the three record types conform.

5.2.1.3 String Conversions

String conversion functions are used primarily to right-extend a string with 'fill' or 'padding' characters of the user's choice.

$string(<length>,<string expression>[,<fill>])

$<string type identifier>(<string expression>[,<fill>])

where

<length> ::= <positive integer expression>
<fill> ::= <character expression>

These functions return a fixed string whose length is specified by the length given as argument or by the length of the target type. The value returned is obtained from the string expression used as the argument, right-truncated or right-extended (by the fill character) as required by the length of the result. If no fill is specified, the blank character is used.

5.3 FILE_VARIABLE_CONSTRUCTORS

Indefinite file variable constructors can be used only for initializing file variables (cf. 6.7) or as the arguments of definite file variable constructors. The latter can be used for assignment operations involving file variables of the same types.

```
<indefinite file variable constructor> ::= [<file spec>]

<definite file variable constructor> ::=
     $<file type> [<file expression>]

<file spec> ::= <file attributes>  "cf.  6.7"

<file attributes> ::= <file attribute> {,<file attribute>}
<file attribute> ::= <old or new>
                   | <mode>
                   | <encoding>
                   | <position>
                   | <actual file name>
                   | <page size>
                   | <end of page proc>

<old or new> ::= #old | #new
<mode> ::= #in[,#out] | #out[ ,#in]
<position> ::= #first | #asis | #last
<actual file name> ::= <string expression>

<page size> ::= #pagesize (<number of lines>)

<end of page proc> ::= #pageproc (<procedure reference>)

<encoding> ::= #codeset (<codeset>)

<codeset> ::= 'ascii'
            | 'ebcdic'
            | 'ascii63'
            | 'ascii612'
            | 'native'
            | <others as required>
```

See section 6.7 for a complete coverage of file attributes.

6.0 VARIABLES, SEGMENTS, AND FILES

6.0 VARIABLES, SEGMENTS, AND FILES

6.1 VARIABLES AND VARIABLE DECLARATIONS

Variables take on values of a subset of the SWL types: fixed or variable bound types, adaptable types, and bound variant record types.

Variables of fixed or variable bound type can be declared by an explicit variable declaration (see below) or can be declared as formal parameters of procedures (cf. 8.1).

Variables of adaptable or bound variant record type can only be declared as formal parameters of procedures, and must otherwise be explicitly established by storage management operations (cf. 10.4).

6.1.1 ESTABLISHING VARIABLES

This process involves:

a) The determination of the type of the variable;

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

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

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

Explicitly declared variables are automatically established on each entry to the block (cf. 7.5) in which they were declared. However, so-called 'static' variables (cf. 6.2.2) are established once and only once.

Formal parameters of procedures are automatically established on each call of that procedure. If the procedure is associated with a coproc, establishment occurs on each creation of that coproc.

6.0 VARIABLES, SEGMENTS, AND FILES

6.1.1 ESTABLISHING VARIABLES

So-called 'allocated' variables are explicitly established by storage management operations (cf. 10.4) (for type determination and storage allocation), and by assignment operations (for initialization).

6.1.2 TYPING OF VARIABLES

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

6.1.2.1 Instantaneous Types

The type to which a variable is fixed at a specific time during execution of a program is called its instantaneous type (at that time). It is a variable's instantaneous type that is actually used to determine the operations it may enter into at any point in time. In general, two variables whose instantaneous types are equivalent can enter into dyadic operations defined for that type.

The instantaneous type of variables is fixed in the following manner:

1. Types themselves are fixed on entry to the block in which they are declared (by an explicit type declaration), and remain fixed until exit from that block.

2. The instantaneous type of declared variables and formal parameters (of fixed or variable bound type) is determined as follows:

a) If their type is specified solely by a type identifier, the type is fixed on each entry to the block containing the declaration of the identified type, and remains fixed until exit from that block.

b) If their type is specified by an explicitly spelled out type, then the type is fixed on each entry to the block containing the variable or procedure declaration, and remains fixed until exit from that block.

3. Variable bound parts (if any) of adaptable and bound variant record types are fixed as above for variable bound types.

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

a) Formal parameters of such types are fixed by the instantaneous types of their corresponding actual parameters on each call on the procedure of which they are a part. (See Section 10.3.1 for the rules for fixing parameters.) If the procedure is associated with a coproc, the type is fixed on each creation of that coproc.

b) Explicitly allocated variables of such types are fixed by the allocation operation. See Section 10.4.1 for the rules for fixing such variables.

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

Example:

```
var j, m, n : integer;
    .
    .
    .
begin "first block"
    type t1 = string (n) of char;
      .
      .
      .
    begin "second block"

      var S : t1,
          A1 : array [j] of t1,
          A2 : array [j] of string (m) of char;
      proc P (val A : array (*) of t1);
          .
          .
          .
      procend;
          .
          .
          .
      P(A1);
          .
          .
          .
```

6.0 VARIABLES, SEGMENTS, AND FILES

6.1.2.1 Instantaneous Types

The type, t1, is fixed on each entry to the first block, thus fixing the instantaneous type of the variable, S, and the component type (but only the component type) of the variable, A1, and the formal parameter, A. The instantaneous types of the variables, A1 and A2, are fixed on each entry to the second block. The instantaneous type of the formal parameter, A, is fixed to that of the variable A1 on each procedure call, P(A1).

6.1.3 EXPLICIT VARIABLE DECLARATIONS

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

```
<variable declaration> ::=
    var [<file or variable spec>{,<file or variable spec>}]

<file or variable spec> ::= <variable spec>
                          | <file variable spec> "cf.  6.7"
                          | <empty>

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

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

<variable identifier> ::= <identifier>

<alias> ::= alias ' <alphabet> { <alphabet> } '
     "cf.  7.7.1 for semantics of alias"
```

6.2 ATTRIBUTES

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

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

<scope or storage attribute> ::=
                    [<scope attribute> [,<storage attribute>]]
                  | [<storage attribute> [,<scope attribute>]]
```

6.2.1 ACCESS ATTRIBUTE

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

The read and write attributes can be associated with variables (and segments, cf. 6.5) to specify whether values can be read or written-over.

The execute attribute cannot be associated with variables, is automatically associated with procedures and labels, and is otherwise used solely to declare segments (cf. Section 6.5) containing procedures. The degree of support for the execute attribute will be a function of the link-loader (link-editor) and the operating system, so will be system-dependent.

Variables can be declared with either the read attribute or with no attributes at all. In the former case, the variable is called a 'read-only variable;' in the latter case, both the read and the write attributes are automatically associated with the variable, which is then called a 'read-write variable.'

Read-write variables can be used freely for purposes of retrieval and reassignment: in expressions (cf. Section 9), as objects of assignment (cf. Section 10.1) and as actual parameters of procedures (cf. Section 10.3.1).

Read-only variables can (and should) be initialized (cf. Section 6.3), may not be used as objects of assignment, and may be used as actual parameters only if the corresponding formal parameter is either a val parameter or a read-only ref parameter (cf. Sections 4.6 and 8.1.2).

Examples:

```
var     v1 : [#read] integer := 10; "v1 is read only, but
                                     initialization is valid"
var     v2 : real ; "v2 may be 'read' and 'written'"
```

6.2.2 STORAGE ATTRIBUTES AND LIFETIMES

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

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

is freed is called the 'lifetime' of the variable.

6.2.2.1 Automatic Variables

The lifetime of an 'automatic variable' is the block (cf. Section 7.5, Blocks) in which it was declared: allocation and initialization occur on each entry to that block and freeing occurs on each exit from that block.

6.2.2.2 Static Variables

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

6.2.2.3 Lifetime Conventions

If neither storage attributes nor scope attributes (cf. Section 6.2.3) are specified, then the variable is treated as an automatic variable.

If the static attribute is specified or if a segment is specified (cf. Section 6.5) then the variable is treated as a static variable.

If any of the scope attributes (cf. Section 6.2.3) are specified, then the variable is treated as a static variable.

Variables of variable-bound type (cf. Section 4.1) cannot be static variables.

Variables declared at the outermost level of a compilation unit (cf. 7.1) are treated as static variables.

6.2.2.4 Lifetime of Formal Parameters

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

6.2.2.5 Lifetime_of_Allocated_Variables

Allocated variables are established (but not initialized) by an explicit allocation operation, and become undefined when they are explicitly freed or their containing storage variable ceases to exist.

6.2.2.6 Pointer_Lifetimes

Warning: Note that generally a pointer value has a finite lifetime 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 their containing storage variable ceases to exist. Attempts to reference non-existent variables by a designator beyond their lifetime is a programming error and could lead to disastrous results.

6.2.3 SCOPE ATTRIBUTES

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

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

6.0 VARIABLES, SEGMENTS, AND FILES
6.2.3 SCOPE ATTRIBUTES

Neither xref nor external variables can be initialized, and each carries the de-facto static storage attribute.

There should exist only one declaration of a given variable (identifier) with the xdcl attribute within a compilation unit (cf. Section 7.7) or within a group of compilation units to be combined for execution.

If a variable declaration contains either the xref or external attribute, then it may not also contain a segment identifier attribute.

6.2.4 FILE ATTRIBUTES

See section 6.7 .

6.3 INITIALIZATION

Initializations are used to specify values to be assigned to explicitly declared variables each time such variables are established.

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

<initialization expression> ::= <expression>
| <indefinite value constructor>

Whenever the variable is established (cf. 6.1.1), the type of the variable is determined, storage for a variable of that type is allocated, the initialization expression is evaluated, and the resultant value is assigned to the variable according to the normal rules for assignment (cf. 10.1).

6.3.1 INITIALIZATION CONSTRAINTS

1. If no initialization is specified, the initial value is undefined.

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

3. An asterisk, '*', can be used in indefinite value

constructors to indicate uninitialized elements of arrays and records. The initial values of such uninitialized elements are undefined.

4. Initialization expressions may not contain references to values of variables declared in the same block as that containing the initialization. However, references to values of pointers to variables, procedures and labels declared in the same block are allowed.

More precisely, if 'E' identifies a variable, procedure or label declared in the block containing the initialization, then its use in a factor of the initialization expression is constrained to the factor '^E'.

6.3.2 FILE VARIABLE INITIALIZATION

See section 6.7, as special initialization rules apply for file variables.

6.4 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 sharing common access attributes. 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] and [#read, #write]. Support of the #execute attribute will be system-dependent, related to both link-loaders and the operating systems themselves.

Note that SWL segments are primarily designed to group things together, and have no a-priori relationship (or lack of one) to hardware supported segments.

A segment identifier may never be a prong (cf. 7.2, Modules).

6.5 VALID COMBINATIONS OF ATTRIBUTES AND INITIALIZATIONS

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

	ATTRIBUTE	INITIALIZATION	SAME AS
(1)	none	optional	
(2)	#read	required	
(3)	static	optional	
(4)	static,#read	required	
(5)	xdcl	optional	(7)
(6)	xdcl,#read	required	(8)
(7)	xdcl,static	optional	(5)
(8)	xdcl,static,#read	required	(6)
(9)	xref	prohibited	(11)
(10)	xref,#read	prohibited	(12)
(11)	xref,static	prohibited	(9)
(12)	xref,static,#read	prohibited	(10)
(13)	external	prohibited	
(14)	external,#read	prohibited	
(15)	* <segment id>	optional	
(16)	* <segment id>,#read	required	
(17)	* <segment id>,xdcl	optional	
(18)	* <segment id>,xdcl,#read	required	

(*) Static is implied for segments.

6.0 VARIABLES, SEGMENTS, AND FILES
6.5 VALID COMBINATIONS OF ATTRIBUTES AND INITIALIZATIONS

Examples:

```
segment s1 : [#read, #write], s2 : [#read], s3: [#execute];
kar: [#read,xdcl,s1] integer := 27; "correct declaration"
kbr: [xdcl,s2] real;
    "improper declaration -- access attribute of
     kb is read-write, while s2 is read only"
ker: [s3] integer;
    "improper declaration.  -- no accesses allowed
     to segment s3"
kdr: [s1] integer ; "correct declaration"
ker: [s2, #read, xdcl ] real;
    "improper--initialization required
                for xdcl, read-only"
pir: [static,xdcl,#read] real := 3.14159265 ;
    "correct declaration and initial value assignment."

                          " * * * * "

seglonseg2 : [#read, #write, #execute] ;
good : [seg2, #read] integer := 63 ; "correct declaration"
bad : [seg2,xref] boolean ;
    "improper use of both a segment identifier and
      xref attributes in same variable spec"
conv : [xdcl] real := 39.37 ;
    "correct declaration with static attribute implied"

v3r: [xref,#read] boolean ; "correct declaration of v3"
v4r: [#read,external] integer; "correct use of
      access attribute with external scope attribute"
v5r: [#read, xref] real := 2.54; "error because initial
      value assignment not allowed with xref attribute."
v6r: [external, #read ] boolean := true;
    "error, initial value assignment never allowed
      with external attribute."
```

6.6 VARIABLE REFERENCES

```
<variable> ::= <variable reference>

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

6.6.1 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 pointer reference and the notation

<pointer reference>^

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

<pointer reference>^^

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

Given a variable identifier, the notation to obtain a pointer value to the variable is:

^<variable identifier>

However, successive applications of the up arrow for such purposes is not allowed (See Evaluation of Factors, 9.1).

Pointers are always bound to a specific type (cf. Section 4.2.3) and pointer variables may assume, as values, only pointers to objects of that type.

The special value nil is used to denote that a pointer variable has no current assignment to a location. Note the use of the typed nil value constructor for use when a typed pointer is required, such as in the assignment to a union variable. (cf. 5.1.2 and 5.1.3 for value constructors.)

6.6.1.1 Examples_of_Direct_Pointer_References

```
var i, j, k : integer, "integer variables"

    pi : ^integer, "pointer variable of type: pointer to integer"

    ppi : ^^integer, "pointer variable of type:
                      pointer to pointer to integer"
```

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

i := 10 ; "the integer variable i is given the value 10"

pi := ^i ; "the pointer variable pi takes on the value:
            pointer to integer variable i"

ppi := ^pi ; "the pointer variable ppi takes on the value:
              pointer to pointer variable pi"

ppi := ^^i ; "not permitted--^^ ...  ^<identifier>
              is not an allowable expression"

j := pi^ ; "the integer variable j takes on the value
            of the integer variable i"

k := ppi^^ ; "the integer variable k takes on the
              value of the integer variable i"

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

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

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

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

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

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

6.6.2 SUBSTRING REFERENCES

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

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

<first char> ::= <positive integer expression>
<substring length> ::= <positive integer expression>
                     | *
```

Values of string variables are ordered n-tuples of character

values (or the null string). Substring references yield fixed strings defined as follows.

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

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

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

b) 'S(i,k)' yields a fixed string of length k, consisting of the i-th through the (i+k-1)-th character of S;

c) 'S(i,*)' is equivalent to 'S(i,n-i+1)'.

Otherwise, an error results.

Example:

If a string variable is declared and initialized 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
```

If a pointer variable is then declared and initialized by:

```
var ps : ^string (6) of char := ^S;
```

then

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

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

```
const str24 = 'helper';

    ...

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

6.6.3 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. A subscript may be of any type that can be assigned to a variable of the corresponding index type. Note that, to this end, any subrange is considered to be of the same type as its parent range (or any subrange thereof).

Example:

If an array variable is declared and initialized by

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

and an integer variable is declared and initialized by

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

then the following relations hold

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

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

If a pointer variable is then declared and initialized by:

```
var pA : ^array [1..5] of integer := ^A;
```

then

pA^[i] becomes a valid reference to components of A.

Equivalence of Dimensionality of Arrays

If the components of an array are a second array, then this is

fully equivalent to a two-dimensional array (cf 4.3.3, Array Type). Both declarations given in the following are equivalent

```
var vect : array [1..10] of array [1..16] of real
var vect : array [1..10,1..16] of real
```

and both of the following expressions for denoting a component of the array are equivalent:

```
vect[ 10,14]
vect[10][14]
```

6.6.4 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 itself be of record type, in which case

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

becomes a valid field reference.

6.0 VARIABLES, SEGMENTS, AND FILES
6.6.4 FIELD REFERENCES

Example:

For the record variable declared and initialized by

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

var R : tR := [23,false,true,[3,5,73]];
```

the following relations hold

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

If a pointer variable is then declared and initialized by:

```
var pR : ^tR := ^R;
```

then

```
pR^.age, pR^.married, ...
```

become valid references to fields of R.

6.6.5 ADAPTABLE AND BOUND VARIANT REFERENCES

Adaptable and bound variant record types can be used as formal parameters of procedures, in which case they are referenced and treated as variables. In all other cases they must be addressed indirectly through pointers that are generally produced as by-products of allocation operations used to type-fix and allocate storage for variables of such types.

The notation

```
<pointer reference>^
```

is used to reference such variables, and can be used as a variable reference (cf. 6.6.1). For example:

```
type R = record "adaptable record type"
             f1 : integer,
             f2 : string(*),   "adaptable field"
         recend;

var pR : ^R := nil;  "pointer to adaptable"

allocate pR : [10];  "allocate fixed instance of R"

    pR^ := [100,'HENRY'];  "initial value assigned"

pR^.f1 := 2 * pR^.f1;  "change value of first field"
```

6.7 FILE_VARIABLES

An actual file is accessed through a file variable associated with that file. The association is effected by an open statement, and de-association is effected by a close statement (cf. 10.5.2). File variables take on as values an undefined record structure whose component values describe the kind and current state of the associated actual file in terms of the actual file type (cf. 4.8), a file specification, and other pertinent information.

```
<file variable> ::= <variable>
```

Warning: If a file variable is assigned to another file variable, and either used for manipulation of a file, then resulting changes in the file variable used are not implicitely reflected in the other file variable. The use of both file variables for simultaneous manipulation of the same file could result in serious errors.

6.7.1 FILE SPECIFICATION

A file variable is declared, by a file variable spec, in terms of an identifier, a file type, and an optional set of initial file attributes.

```
<file variable spec> ::= <variable identifiers> :
             [<scope or storage attributes>]
             <file type>    "cf. 4.8"
             [<file variable initialization>]
```

A file spec consists of an optional set of non-repeated attributes, and is used to initialize file variables, to specify

6.0 VARIABLES, SEGMENTS, AND FILES
6.7.1 FILE SPECIFICATION

(or respecify) file attributes in open statements (cf. 10.5.1), and to spell-out file-variable constructors (cf. 5.2.3).

```
<file spec> ::= <file attributes>
```

6.7.1.1 File Attributes

```
<file attributes> ::= <file attribute> {,<file attribute>}
<file attribute> ::= <old or new>
                   | <mode>
                   | <encoding>
                   | <position>
                   | <actual file name>
                   | <page size>
                   | <end of page proc>

<old or new> ::= #old | #new
<mode> ::= #in[,#out] | #out[ ,#in]
<position> ::= #first | #asis | #last
<actual file name> ::= <string expression>

<page size> ::= #pagesize (<number of lines>)

<end of page proc> ::= #pageproc (<procedure reference>)

<encoding> ::= #codeset (<codeset>)

<codeset> ::= 'ascii'
            | 'ebcdic'
            | 'ascii63'
            | 'ascii612'
            | 'native'
            | <others as required>
```

The #old attribute indicates that the variable is to be associated with an existing file. The #new attribute indicates that a file must be created before such an association can be made. The creation of new files is effected by the open procedure.

The mode attribute specifies whether the file associated with the variable is to be read, written, or both. Direct files, legible files, and binary files may have both #in and #out for <mode> simultaneously; but print files may have only #out for <mode>.

The encoding attribute specifies the external representation of a file (its so-called 'codeset'), and implicitely indicates

the conversion process that is invoked during reading and writing of the file. An initial collection of codesets, corresponding to existing representations, is provided. The <codeset> value 'native' will select the standard codeset for the machine on which execution is to occur.

The position attribute permits the open procedure to position the file at its beginning, its end, or some existing point. (The use of the #asis attribute and its effects will be operating system dependent.)

The actual_file_name is used solely to identify the actual file to a host operating system. Its lexical formation rules may be system-dependent.

The #pagesize and #pageproc attributes are associated with print files only, and define the size of a page and the procedure to be activated when a page change is ready to occur.

The end-of-page_procedure is a user-defined procedure that is called whenever the current line number for the print file exceeds the specified page size for the file. It is responsible for issuing a conventional page eject (cf. 10.5.5.2, <eject statement>), and its parameter list is assumed to be specified as follows:

(ref <file identifier> : <print file type>;

val <integer identifier> : <integer type> "next page no.")

If no end-of-page procedure has been specified for the file, a conventional page eject is issued and the line number is set to one (1).

The user may set, or reset, these attributes directly through the following procedures:

#setpagesize(<print file variable>,<number of lines>)

<number of lines> ::= <integer expression>

#setpageproc(<print file variable>,<procedure reference>)

and may interrogate the current page size through the function #curpagesize (<print file variable>) .

6.7.2 FILE VARIABLE INITIALIZATION

File variable initializations are used to specify one or more file attributes when declaring file variables.

```
<file variable initialization> ::=
        := <file expression>
      | := <indefinite file variable constructor>

<indefinite file variable constructor> ::= [<file spec>]

<file spec> ::= <file attributes>
```

The actual file need not be completely specified with the declaration of the file variable. Any or all of the file attributes, including the actual file name, can be specified or re-specified with the open statement used to associate an actual file with the file variable.

7.0 PROGRAM STRUCTURE

7.0 PROGRAM STRUCTURE

7.1 COMPILATION UNITS

A SWL program is a collection of *statements* (cf. Section 10) and *declarations* (cf. Section 7.3) which is meant to be translated, via a *compilation* process, into a SWL *object module*. Object modules resulting from separate compilations can be combined, via a *linking* process, into a single object module, and may undergo further transformations into a form capable of direct execution by members of the IPL line.

The collection of statements and declarations may also include *compile-time directives* (cf. Section 12) which are used solely to construct the program being compiled and to otherwise control the compilation process, rather than having any meaning in the program itself. The result of processing the collection according to these directives must be one or more valid *compilation units*, which are distinguishable cases of a *module*.

<compilation unit> ::= <module declaration> "cf. Section 7.2"

Since statements are constrained to appear solely within the body of a *procedure declaration* (cf. Section 8.1), compilation units consist solely of a list of declarations. All such declarations must be capable of being evaluated (cf. Section 7.9) at the time of compilation. All variables declared in a compilation unit's declaration list will automatically be given the *static storage attribute* (cf. Section 6.2.2).

7.2 MODULES

A module is a collection of declarations packaged so as to make visible the identity of those objects declared within the module which are meant to be shared with other parts of the same compilation unit or with separate compilation units. A module is introduced by a module declaration.

7.0 PROGRAM STRUCTURE
7.2 MODULES

```
<module declaration> ::=
        module [<module identifier>] [(<prongs>)];
            <module body>
        modend [<module identifier>]

    <module identifier> ::= <identifier>[<alias>]
        "cf. 7.7.1 for semantics of alias."
    <prongs> ::= <identifier list>
    <module body> ::= <declaration list>

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

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

Prongs are identifiers declared in the body of the module, and - together with module packaging itself - are used solely to control the scope of identifiers (cf. Section 7.3 thru 7.7)

7.3 DECLARATIONS AND SCOPE OF IDENTIFIERS

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

```
<declaration> ::= <type declaration>        (cf. Section 4.1)
                | <constant declaration     (cf. Section 5.1)
                | <variable declaration>    (cf. Section 6.1)
                | <segment declaration>     (cf. Section 6.4)
                | <module declaration>      (cf. Section 7.2)
                | <procedure declaration>   (cf. section 8.1)
                | <label declaration>       (cf. Section 8.3)
                | <empty>
```

The programmatic domain over which all uses of an identifier are associated with the same object is called the scope of the identifier. Within a compilation unit, such a programmatic domain is either a module body (cf. Section 7.2) or a block body (cf. Section 7.5). In the former case, the scope is a declaration list; in the latter, a statement list preceded by an optional declaration list.

The scope of an identifier is determined by the context in which it was declared and by optional scope attributes (cf. Sections 6.2.3, 8.1 and 7.7) which may be associated with declarations of variables and procedures.

7.4 MODULE - STRUCTURED SCOPE RULES

Modules are static constructs designed solely to control the scope of identifiers according to the following rules:

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

2. An identifier whose scope is the body of a module, and is also listed as a prong of that module has its scope extended 'outward' to include the body of the module or block which includes the module declaration as one of its constituent declarations.

3. Identifiers (of variables and procedures) whose scope is the body of a compilation unit, and are also listed as prongs of that compilation unit, have their scope extended 'outward' to include object programs resulting from other compilation units. Scope attributes (cf. Section 7.7) may also be used to specify such extensions of scope for variables and procedures. The prongs of a compilation unit are constrained to identify only variables and procedures, and the xdcl scope attribute is automatically given to any variable or procedure whose identifier is listed as the prong of a compilation unit, unless it is explicitly declared with one of the other scope attributes.

7.5 BLOCKS

A block is either a begin statement (cf. Section 10.2.1) or a procedure declaration (cf. Section 8). A block body consists of a statement list preceded by an optional declaration list. Blocks have three functions:

1. Like modules, blocks control the scope of identifiers.

2. Unlike modules, blocks control the processing of declarations and determine when declarations take effect (cf. Section 7.9)

3. Unlike modules, blocks include statements, which translate into algorithmic actions in the resulting program.

7.6 BLOCK - STRUCTURED SCOPE RULES

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

2. If an identifier labels (cf. Section 10) a statement of the constituent statement list of a block, then its scope is the body of that block. However, if an identifier labels a statement of one of the constituent statement lists of other structured statements (cf. Section 10.2), then its scope is restricted to that statement list.

3. If the scope of an identifier includes a block, then its scope is extended 'downward' to include the body of that block, unless the body includes a re-declaration of the identifier.

4. Field selectors are identifiers introduced as part of the declaration of a record type (cf. Section 4.3.4) for purposes of selecting fields of records (cf. Section 6.4.4). Except for the restriction that field selectors associated with the same record type must be unique, identifiers used as field selectors may be re-declared with impunity.

5. Except for field selectors, no more than one declaration of an identifier can be included in the constituent declarations and statements of a block's body.

7.7 SCOPE ATTRIBUTES

The scope attributes xdcl, xref and external (cf. 6.2.3) cause the scope of identifiers to be extended, in a discontinous manner, to include other compilation units, but do not otherwise contravene either module-structured or block-structured scope rules.

Variables and procedures that are part of one compilation unit, but are meant to be referenced from other compilation units, must have the xdcl attribute associated with them either by explicit declaration or by virtue of having their identifiers listed as a prong of the compilation unit. Other compilation units which are meant to reference such objects must declare them with the xref attribute.

Variables, but not procedures, may also be declared with the external attribute, which is intended to permit SWL programs and programs written in other languages to share data. Variables

with the external attribute may be referenced in any compilation unit in which they are declared.

Neither xref nor external variables can be initialized (cf. Section 6.4), and all xdcl, xref and external variables are automatically given the static storage attribute (cf. Section 6.2.2)

The declarations for objects shared among compilation units must match; for example, an identifier with the xdcl attribute in one compilation unit and the xref attribute in other compilation units must denote tne same object in all such compilation units. Violations of such matching rules are detected during the linking process.

7.7.1 ALIAS NAMES

An 'alias' is an alternate spelling which may be specified for an identifier. Its reasons for existence are varied: to meet system-requirements of spelling which are invalid in SWL, to equate two differing spellings for an entity between two different compilation units, to avoid identifier spelling conflicts among different compilation units or with system standard names, etc.

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

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

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

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

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

7.0 PROGRAM STRUCTURE
7.7.1 ALIAS NAMES

Examples:

```
module outer alias 'PT*T4*' (x,a); ...

proc [xdcl] searcher alias 'util*221' (ref ist2,...

var V2 alias 'flag2', V3 alias 'T##3.1B' : [xdcl ] integer;
```

7.8 EXAMPLES OF SCOPE RULES

```
module outer(x, a) ;          "scope of x and a is extended
                               outward to the body of module
                               or block including this
                               declaration"

 block1: begin
         var x, y, z ; integer ; ...  ;

         x := y + z ;          "block1's x, y, and z"
         y := a + b ;          "block1's y, a, and b (see inner
                                module below)"
         z := p + z ;          "Invalid: not within scope of p
                                (see block2 below)"
           ...  ;
         block2:
                 begin
                 var x, y : integer ;
                                "Valid redeclaration of x, y"
                 var p : [xdcl] integer,
                                "scope of p extended to other
                                 compilation units, but not"
                                 extended to include block1"
                       q : [xref] integer ; "q from different
                                             compilation unit"
                 x := y + z ; "z of block1 and x and y of block2"
                   ...  ;
         end block2 ;
           ...  ;              "Now back in block1"
         module inner(a, b) ;"Still in block1, but within
                               inner module"
                 var a, b : integer ;  "a, b are within the scope
                                of block1, since they are pronged"
                 var z : integer ;      "Valid redeclaration of z,
                                         since it is not pronged"
                   ...  ;     "Other declarations of inner module"
         modend inner
```

7.0 PROGRAM STRUCTURE
7.8 EXAMPLES OF SCOPE RULES

```
        end block1 ;
 block3: begin
        var x, y, z : integer ;
                "Valid redeclaration of x, y, and z"
          ... ;
        end block3 ;
modend outer ;

                        " * * * * * * "

L1: if x < y then ... ;
        .
        .
        .
        L2:  x := z / y ; ... ; goto L2 ; ... ;
          "Valid: L2 can be reached from within
            this statement list"
   orif x > z then ... ;
          .
          .
          .
          x := y / z ; ... ; goto L1 ; ... ;
         "Valid: Scope of L1 is entire block
          containing the if statement"
          .
          .
          .
   else y := y - a ; ... ; goto L2 ; ... ;
       "Invalid: L2 cannot be reached from this statement list"
   ifend
```

7.9 DECLARATION PROCESSING

7.9.1 BLOCK-EMBEDDED DECLARATIONS

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

Block-structure also governs declaration processing during execution of the resulting programs. On entry to a block, all declarations included in the block's constituent list are again collected together, evaluation of declarations that could not be completely evaluated during compilation is completed, storage for

automatic variables (cf. Section 6.2.2.1) is allocated, and all identifiers declared by such declarations become accessible. On exit from a block, all identifiers declared within that block become inaccessible and the values of automatic variables become undefined.

7.9.2 COMPILATION-UNIT--EMBEDDED DECLARATIONS

Objects declared in the body of a compilation unit (cf. Section 7.1) are associated with no block at all. Such declarations must be evaluated, and required storage allocated, prior to program execution. Accordingly, all variables so declared are automatically given the static storage attribute (cf. Section 6.2.2), as are all scope-attributed variables (cf. Sections 7.7 and 6.2.3), and all such variables are constrained to be of fixed-bound type (cf. Section 4.2) to ensure that their types can be evaluated during compilation. In addition, formal parameters of procedures declared in a compilation unit's declaration list may not be of variabe-bound type (cf. Section 4.2), since the evaluation of a procedure declaration involves the evaluation of the types of its formal parameters (cf. Section 8.1).

7.9.3 ORDER OF EVALUATION OF DECLARATIONS

Apart from the above rules, no specific order of evaluation of declarations is defined, nor is the order of evaluation of expressions entering into such declarations defined. Thus, care must be exercised in spelling-out declarations. For example:

```
var L : integer, U : union (integer, ...);

  L := 5; U := 3;

  begin
       var A : array [1..L :=: U, 1..L] of ...;
       ...
  end
```

In the above example, the integer variable L has been assigned the value 5 and the union variable U has been assigned the value (integer, 3) prior to entry to the begin block. On entry to the block, evaluation of the index ranges of the array A must be done. This evaluation can result in either A[1..3, 1..3] or A[1..3, 1..5], depending on which indicial expression is

evaluated first (cf. Section 9.2.1 for the value conformity operator ':=:' which, in this case, would assign L the value 3). Since no precise order of evaluation is guaranteed, the result of the above program fragment is undefined.

8.0 PROCS, COPROCS, AND LABELS

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, by a return statement, or by an exit statement). The results of returning by any means from a functional procedure prior to assignment of a value to the function designator (for the current execution) are undefined.

A procedure call statement (cf. 10.3.1) 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 following the procedure call statement.

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 a single thread of control being passed back and forth amongst them through the resume statement.

The create statement (cf. section 10.3.2) creates the necessary environment for the execution of a procedure as a coprocess. 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

There are two forms of procedure declaration:

```
<procedure declaration> ::=
    proc [ xref ] <proc spec>
  | proc[[<proc attributes>]]<proc spec>;<proc body><proc end>

<proc attributes> ::= <proc attribute>{ ,<proc attribute>}
<proc attribute> ::= xdcl | reedep | <segment identifier> | main

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

<alias> ::= alias ' <alphabet> { <alphabet> } '
               "cf.  7.7.1 for semantics of alias"

<proc type spec> ::=
    [[<proc type attributes>]][<parameter list>][<return type>]

<proc type attributes> ::=
      <null construct (for expansion purposes)>

<parameter list> ::= (<param segment>{;<param segment>})
<param segment> ::= <reference params>|<value params>

<reference params> ::=
     ref <reference param> { ,<reference param> }

<reference param> ::=
      <formal param list> : [[ #read ]] <ref type>

<value params> ::= val <value param>{ ,<value param>}

<value param> ::=
      <formal param list> : [[ #read ]]<val type>

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

<ref type> ::= <SWL type>
<val type> ::=
     <type> | <adaptable type> | <bound variant record type>

<return type> ::= <basic type>

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

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

8.0 PROCS, COPROCS, AND LABELS

8.1 PROCEDURE DECLARATIONS

<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 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.

Outermost level procedures, i.e., those whose declarations are not contained in another procedure, must have a fixed type determined at compile time. Thus, none of their 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). Formal parameters of outermost level procedures may be of either fixed bound type or adaptable bound type.

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 | reqdep | <segment identifier> | main

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 reqdep specifies that the procedure is potentially representation dependent and gives permission for the use of those portions of the language that are representation

8.0 PROCS, COPROCS, AND LABELS
8.1.1 PROC ATTRIBUTES

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 carrying the same segment identifier.

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

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

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> ::= (<param segment>{;<param segment>})
<param segment> ::= <reference params>¦<value params>

<reference params> ::=
     ref <reference param> { ,<reference param> }

<reference param> ::=
      <formal param list> : [[ #read ]] <ref type>

<value params> ::= val <value param>{ ,<value param>}

<value param> ::=
      <formal param list> : [[ #read ]] <val type>

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

<ref type> ::= <SWL type>
<val type> ::=
     <type> ¦ <adaptable type> ¦ <bound variant record type>
```

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 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. See Section 10.3.1 for precise rules governing call-by-value parameter passing.

The type of a formal call-by-value parameter may be any data type, adaptable type or bound variant record type except for the so-called non-value types: heaps; arrays or stacks of non-value types; and records containing a field of a non-value type.

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. See Section 10.3.2 for precise rules governing call-by-ref parameters.

The type of a formal call-by-reference parameter may be any SWL type.

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

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. The return type must be specified in the proc type spec for any procedure which is a functional procedure (cf. 4.6.2); and an assignment statement (at least one) to the procedure identifier must occur within the procedure body.

<return type> ::= <basic type>

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

8.0 PROCS, COPROCS, AND LABELS
8.1.3 FUNCTIONS AND RETURN TYPE

A function may neither be invoked by a procedure call statement nor used as a coproc. (It may however lie within the dynamic execution of a coproc, and there is no restriction against its containing a resume statement.)

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.
     This could have been written as a
     functional procedure."

     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)"

           d := 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
```

8.0 PROCS, COPROCS, AND LABELS
8.1.3 FUNCTIONS AND RETURN TYPE

```
                    " * * * * * * "

        "functional procedure Finder"

  proc Finder(val wanted: ident,
       ref table : array [0..47] of entries : retype) ^retype ;

  var k : integer ;

    for k := 0 to 47 do "trivial search"

      if table[k].fn = wanted then
        Finder := ^table[k]
       .return
      else
        Finder := nil
      ifend

    forend

  procend ; "Finder either set to point
             to table entry or set to nil"
    ...
   k22 := k22 + Finder^.kstep ;
```

8.2 COPROCS

A coproc is created by execution of the create (cf.10.3.2) statement:

create (<pointer to coproc>, <procedure call statement>);

where the coproc reference serves as a linking mechanism between (among) two (or more) synchronous coprocs in execution.

Any declared procedure may be designated as a coproc in a create statement.

Execution of the create statement does not cause execution of the designated procedure (in the procedure call statement part) to commence. It does cause the creation of a separate environment (for procedure entries, parameter passing, and procedure exits), and the passing of actual parameters, including those called by value.

Execution of the procedure body of the coproc begins with the execution of a resume command using the coproc reference (cf. 10.3.4). Such execution begins with the declaration list of the

procedure body (at which time automatic variables are allocated), and then with the first statement of the procedure body. Execution of the coproc continues until a resume statement is encountered in execution which returns control to the initializing coproc or alternately causes execution to begin for another coproc or to continue in another coproc.

The form of the coproc resume statement (cf. 10.3.4) is:

resume (<pointer to coproc>^); ...

If such a resume statement occurs dynamically (during program execution) at some point prior to designation by a create statement for the coproc or at some point after the coproc has been destroyed (cf. 10.3.3) then the program is in error. (This is because the environment for the designated procedure has not been--or no longer is--set up as a coproc.)

When an exit is made from a dynamically active coproc by any means other than a resume, then the program is in error. The execution of a resume statement will always transfer control to a different coproc at:

(a) the successor of another resume statement of another coproc, or

(b) the beginning of a specified coproc for which a create has already been done, but for which no resume has yet been done.

A resume statement designating the coproc in which the resume statement occurs is in error.

The main program which initializes the other coprocs with create and resume statements is always implicitly a coproc to those other coprocs. In order to allow it to be resumed by these other coprocs, it must determine its own 'identity' using the #coprocid function (cf. 11.2.13) and assign it to a jointly known variable of type pointer to coproc (see CPPC in example below).

8.3 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.

NOTE: Label declarations are required only in ISWL programs, and should not appear in SWL programs.

9.0 EXPRESSIONS

9.0 EXPRESSIONS

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands (i.e., variables and constants) operators, and functions. Constant expressions (cf. 5.1.1) are expressions involving constants and a subset of the operators and functions.

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

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

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

<power> ::= <factor> | <power> <exponentiation operator> <factor>

<factor> ::= <conformity>|<variable>|<constant>
            |<definite value constructor>|^<variable>|^<label>
            |^<procedure identifier>|<function designator>
            |(<expression>)|<not operator><factor>

<conformity> ::=
      <type identifier> <type test operator> <union variable>
     |<pointer variable> <pointer type test operator>
        <union variable>
     |<variable> <value type test operator> <union variable>

<type identifier> ::= <identifier>|integer|char|boolean|real

<union variable> ::= <variable>

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

<procedure reference> ::= <procedure identifier>
                           |<pointer to procedure>^
```

9.0 EXPRESSIONS

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

<type test operator> ::= ::
<pointer type test operator> ::= :^:
<value type test operator> ::= :=:

<not operator> ::= not
<exponentiating operator> ::= **
<multiplying operator> ::= * | / | mod | and | cand
<sign> ::= + | -
<adding operator> ::= + | - | or | xor | uor
<relational operator> ::= < | <= | > | >= | = | /= | in
```

Examples:

Conformities:
```
real :: basicvar
pint :^: basicvar
int :=: 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)
```

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
```

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

9.0 EXPRESSIONS
9.1 EVALUATION OF FACTORS

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 fixed string (n) value consisting of the characters between the quote marks.

The value of a string when used as a factor in an expression is as follows: if the current length (see following paragraph) of the string is zero, its value is the null string; otherwise, its value is a fixed string of the same current length. In particular, substrings of varying strings are fixed strings. The value of a character when used as a factor in a string expression is a fixed string of length one.

The current_length of a string is defined as follows: The current length of a varying string is defined to be m whenever its value is a fixed string of length m and is defined to be zero whenever its value is the null string. The current length of a fixed string is equal to its length.

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 turn is a constant identifier, the process is repeated until a constant of one of the above forms results. The value is then obtained as above.

The value of a 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.

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.

9.0 EXPRESSIONS

9.1 EVALUATION OF FACTORS

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.

9.2 OPERATORS

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

9.2.1 TYPE TESTING OPERATORS

The type testing operators are used to:

a) determine the type of the value last assigned to a union variable;

b) permit references to that value by concurrently assigning it (or, optionally, a pointer to the value) to the variable (pointer variable) being used as a comparand.

The type being tested-for can be specified by a type identifier, the type of a variable, or the type pointed-to by a pointer variable. These operations are permitted only if a variable of the type being tested-for can be assigned to one and only one member of the union.

9.0 EXPRESSIONS

9.2.1 TYPE TESTING OPERATORS

The type test operator (:!) expects a type identifier on its left, and returns the boolean value true if the type identifier specifies the same type as the type of the current value of the union variable on the right, and false otherwise.

The pointer type test operator (:^:) expects a pointer variable on its left, and returns the boolean value true if the pointer variable is of type pointer to the type of the current 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.

The value type test operator (:=:) expects a variable on its left, and returns the boolean value true if the variable is of the type of the current value of the union variable on the right, but otherwise returns the value false. When the boolean value returned is true, then the variable designated on the left is assigned the value of the union variable. Otherwise the variable designated on the left is unchanged.

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
cand	conditional and true and true = true true and false = false false and false = false false and true = false* *When the first operand is false, the second is never evaluated.	boolean	boolean

Note: The operator cand may not be used for set intersection as and is used.

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 ≡ + a.

The - operator can be applied to integer and real types only. It denotes sign inversion--i.e., -a ≡ 0 - 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. (See also 10.1.1 for the successor assignment statement and predecessor assignment statement, which are analogous to the adding operator.)

9.0 EXPRESSIONS
9.2.5 ADDING OPERATORS

Operator	Operations	Operands	Result
+	addition	integer real	integer real
	concatenation *	string and string string and char	string string
-	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 uor **	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

* The rules for concatenation are as follows. The adding operator, '+', is used as the dyadic concatenation operator. The result of concatenation is a string whose current length is the sum of the current lengths of the two operands and whose value is the string obtained by right-extending the left operand by the right operand.

** If evaluation of the uor (unconditional or) is found to be necessary in the execution of the statement in which it occurs, then the evaluation of both operands is guaranteed to occur. The uor may not be used as a set union operator.

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 and Reals

All six comparison operations < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), = (equal to), and /= (not equal to) are defined between operands of the same scalar type, 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 type boolean the relation false < true defines the ordering.

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

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

9.2.6.2 Comparison_of_Direct_Pointers

Two direct pointers can be compared if they are pointers to either equivalent types or potentially equivalent types. In the latter case, one or both of the pointers may be pointers to variable bound types, adaptables types and bound variant records - whose type must be determined during execution of the SWL program. The instantaneous type of such pointers must be the same as that of the pointer they are being compared with; if it is not, the operation is undefined.

1. Pointers to file type and pointers to control type may be compared for equality and inequality only.

a) Two pointers to file are equal if they designate the same file variable.

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

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

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

2. All six comparison operators are defined for pointers to data type, for adaptable pointers and for pointers to bound variant type.

a) Pointers of such type are equal if they designate the same variable. For pointers to variable bound type, adaptable pointers and bound variant pointers, this means that their instantaneous type (i.e., including current bounds, lengths, or tag fields and variants) must be the same as the type of the pointer they are being compared with.

b) Two pointers with nil value are always equal.

c) The remaining comparison operations are defined as follows. Let ω denote any of the remaining operators <, <=, >=, >, and let p, q, r denote three pointers of equivalent type. If p ω q and q ω r hold true, then p ω r also holds true.

9.2.6.3 Comparison_of_Relative_Pointers

Relative pointer comparison is allowed only for relative pointers of equivalent type. Two relative pointers are

equivalent if they are defined in terms of equivalent object types and equivalent parental types (cf. 4.2.3, Relative Pointer Types). For relative pointers whose object type is a variable bound type, this means that their "instantaneous" object type must be the same as the object type of the relative pointer they are being compared with; if it is not, the operation is undefined.

Comparison of relative pointers is defined as follows:

1. Let p and q denote relative pointers of equivalent type, and let P denote a variable whose type is equivalent to the common parental type of these relative pointers. If the relation #ptr(p,P) = #ptr(q,P) holds (cf. 11.2.15), then p and q are equal.

2. Let p, q, and r denote relative pointers of equivalent type, and let op denote any of the comparison operators <, <=, >=, >. If the relations p op q and q op r both hold, then the relation p op r also holds.

9.2.6.4 Comparison of Strings

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

All six relational operators may be applied to operands whose values are strings; if the current lengths of the strings entering into the operation differ, then the shorter of the two is right-extended with blanks to match the current length of the larger before the operation is carried out. If one of the operands is of type char, it is converted to the string(1) value consisting of that character, and then the rules for unequal length strings are applied if appropriate.

Strings are compared to each other character by character from left to right until total equality or inequality is determined, as follows. Let n be the length of the resulting strings a and b ($n \geq 1$), and op 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 ($1 \leq i < k$)
and a(k) op b(k) ($1 < k \leq n$)

9.2.6.5 Relations Involving Sets

The relation a in S is true if the scalar value a is a member of the set value S. The base type of the set must be the same

as, or a subrange of, the type of the scalar.

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.

9.2.6.6 Relations Involving Arrays and Records

1. Arrays may be compared for equality or inequality only. Two arrays are equal if their instantaneous types are the same (cf. 4.3.3.1) and if elements with corresponding subscript values are equal.

2. Variant records can not be compared. Other record types may be compared for equality or inequality only. Two comparable records are equal if their instantaneous types are the same (cf. 4.3.4.5) and if corresponding fields are equal.

9.2.6.7 Non-Comparable Types

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

9.2.6.8 Table of Comparable Types and Result Types

See the following page.

9.0 EXPRESSIONS
9.2.6.8 Table of Comparable Types and Result Types

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

Operator	Operation	Left Operand	Right Operand	Result
<	- less than	any scalar type T	T	boolean
<=	- less than or equal to	real	real	boolean
>	- greater than	string	string	boolean
>=	- greater than or equal to	string char	char string	boolean boolean
=	- equal to			
/=	- not equal to	any (*) pointer type T	T	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 array type T	T	boolean
		any non-variant record type T	T	boolean
		any pointer type T	T	boolean

(*) Except for pointers to procedure, coproc, label, or file.

9.2.7 EXPONENTIATING OPERATOR

The exponentiating operator ** is defined for an argument which is a positive or negative integer expression, raised to a power which is limited to that of a positive integer expression. It is defined to be left associative, so that a ** b ** c is evaluated as (a ** b) ** c. (This follows the syntactic rules for expressions, cf. 9.0).

9.3 ORDER OF EVALUATION

The rules of composition specify operator precedence according to six classes of operators. The type testing operators have the highest precedence, followed by the not operator, followed by the exponentiating operator, followed by the so-called multiplying operators, then the so-called adding operators, and finally, with the lowest precedence, the relational operators.

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

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

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

10.0 STATEMENTS

10.0 STATEMENTS

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

Statement Labels

A statement may be labeled 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 labeling 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 it must be identical to one of the labels labeling that statement. This is for checking purposes only, and does not affect the meaning of the program.

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

<statement> ::= <unlabeled statement>|<label> : <statement>

<unlabeled statement> ::= <assignment statement>
                        |<structured statement>[<label>]
                        |<control statement>
                        |<storage management statement>
                        |<input-output statement>

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

10.0 STATEMENTS

Example:

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

L1:           x := x + y

L2:L3:L4:     y := z "note multiple labels permitted"
```

Semicolons As Statement Delimiters

Since the successor of the last statement of a statement list is uniquely determined by the structured statement or procedure of which it is a part, semicolons are not required as statement list delimiters. However, since the empty statement (cf. 10.3.9) is allowed, semicolons may be so used for purposes of consistency or presentation.

Examples:

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

L1: x := x + y ;
```

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 to be returned by a function designator.

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

                         |<successor assignment> "cf. 10.1.2"
                         |<predecessor assignment>

                         |<concatenating assignment> "cf. 10.1.3"
```

10.1.1 ASSIGNMENTS TO VARIABLES AND FUNCTIONS

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

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

(a) the so-called non-value types: heaps; arrays or stacks of non-valued types; and records containing a field of non-value type.

(b) any variable or parameter specified as read-only.

(c) any bound variant record.

(d) the tag field of any bound variant record.

The variable on the left (or the return type of the function) and the expression on the right must be of identical instantaneous types (cf. 6.1.2), 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 type of the expression may be one (and only 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 character variable, the expression may be a string whose current length is one or greater. The string value is right truncated to a single character string which is converted to type char and then assigned.

4. If the left part is a fixed string variable, the expression may be a character or a string, and the operation is as follows:

   a) if the expression is a character, it is converted to a fixed string of length one.

   b) if the current length of the string differs from that of the assignee, the string will be either right truncated or right extended with blanks to obtain a string of matching length.

10.0 STATEMENTS
10.1.1 ASSIGNMENTS TO VARIABLES AND FUNCTIONS

c) the assignment is then carried out.

5. If the left part is a varying string variable, the expression may be a character or a string, and the operation is as follows:

a) if the expression is a character, it is converted to a fixed string of length one.

b) if the current length of the string expression exceeds the maxlength of the assignee, the string will be right truncated to the matching current length.

c) the assignment is carried out; the current length of the variable is set to the (possibly revised) current length of the expression.

6. If the left part is a variant record the right part may be a bound variant record of otherwise equivalent type.

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

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

9. If the left part is a stack, the right part must be a stack whose instantaneous component type must be the same as the left part's component type.

10. If the left part is a sequence, the expression may be any sequence.

11. Stacks may be assigned only to stacks of the same component type, and sequences may be assigned only to sequences. After the assignment, source and destination contain the same data values (those stored in the source) and are in identical states with respect to any future operations. If the allocated size of the destination is not large enough to hold the source data, then the program is in error. Data values stored in the destination prior to the assignment become undefined by virtue of having been either overwritten or stored past the space required for the assignment.

12. Warning: Note that generally a pointer value has a finite

10.0 STATEMENTS
10.1.1 ASSIGNMENTS TO VARIABLES AND FUNCTIONS

lifetime (see Section 6.2.2) 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 their containing storage variable ceases to exist. Attempts to reference non-existent variables by a designator beyond their lifetime is a programming error and could lead to disastrous results.

10.1.2 SUCCESSOR AND PREDECESSOR ASSIGNMENT STATEMENTS

These assignment statements furnish the n<u>th</u> successor or n<u>th</u> predecessor of a scalar.

<successor assignment> ::= <scalar variable> :+ <n<u>th</u>>
<predecessor assignment> ::= <scalar variable :- <n<u>th</u>>

where

<n<u>th</u>> ::= <integer expression>

These statements replace the current value of the scalar variable by its n<u>th</u> successor or predecessor, if that successor or predecessor exists. They are equivalent to applying the #<u>succ</u> or #<u>pred</u> function n times to the variable.

If the value of n<u>th</u> is zero then the current value is unchanged. If the value of n<u>th</u> is negative then the program is in error.

If the n<u>th</u> successor or predecessor does not exist, then the program is in error.

10.1.3 CONCATENATING ASSIGNMENT

The concatenating assignment statement is used for purposes of conveniently right-extending varying strings.

<concatenating assignment> ::=

<varying string variable> :+ <string expression>

If the expression is a character, it is converted to a fixed string of length one.

If <u>V</u> denotes a varying string variable and <u>E</u> denotes a string expression, then the concatenating assignment

10.0 STATEMENTS
10.1.3 CONCATENATING ASSIGNMENT

```
V :+ E ;
```

yields the same effect as the variable assignment

```
V := V + (E) ;
```

10.2 STRUCTURED STATEMENTS

Structured statements are constructs composed from statement 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>
                          |<value conformity case statement>
                          |<pointer 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
```

10.0 STATEMENTS
10.2.1 BEGIN STATEMENTS

Example:

```
outer: begin var a, b, c : integer, b2, b3 : boolean ;
     a := r + s ; "r and s declared outside block"
     b := r + 10 ;
L1:  b2 := true ;
inner:
     begin v1, v2 : integer , b2 : boolean ;
          v1 := a + r ;
          v2 := 25 ;
          if v1 < v2 then b3 := true ifend ;
L3:       b2 := false ; "b2 of inner block"
     end inner ;
     if v2 > a then
       a := 1 ; "improper statement, since v2
                 no longer exists"
     ifend ;
     if not b2 then
       b := c
     ifend ; "b2 is true from statement L1.
              the b2 set by statement L3 holds only
              in the block labelled inner."
end outer ;
```

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 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>}
```

10.0 STATEMENTS
10.2.2 IF STATEMENTS

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
```

10.2.3 LOOP STATEMENTS

The loop statement causes unbounded repetition of its component statement list. Thus, exit from a loop statement must be through an explicit change of control.

The successor to the last statement of the constituent statement list of a loop statement is the first statement of the list.

<loop statement> ::= loop <statement list> loopend

10.0 STATEMENTS
10.2.3 LOOP STATEMENTS

Example:

```
type nextid : string (20) of char ;
var scrambloc : array [1..256] of ^field ;
type field = record
               a, b, c : integer,
               d : ^field,
               id : string (20) of char
             record ;
var k : integer, found : boolean ;
     k := 128 ;
     m := k / 2 ;
finder: loop "binary search used"
          if scrambloc[k]^.id = nextid then
            found := true ;
            exit finder
          ifend ;
          if scrambloc[k]^.id < nextid then
            k := k - m
          else
            k := k + m
          ifend ;
          m := m / 2 ;
          if m = 0 then
            found := false ;
            exit finder
          ifend
        loopend ;
        if found then
           ...
        ifend ;
        "the variable found tells whether a string equal to
         nextid is located in the table, the entries of which
         are pointed to by the values in scrambloc. if the
         matching entry is found, then k is the index of the
         entry."
```

10.2.4 WHILE STATEMENTS

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

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

The expression controlling repetition must be of type Boolean. The statement list is repeatedly executed until the expression becomes false. If its value is false at the

beginning, the statement list is not executed at all. The while statement

```
while e do S whilend
```

is equivalent to

```
if e then S; while e do S whilend ifend
```

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

Examples:

```
while a[i] /= x do i := i + 1 whilend

while i > 0 do
   if 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
    "do nothing"
  else
    repeat S until e
  ifend
end
```

10.0 STATEMENTS
10.2.5 REPEAT STATEMENTS

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, 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 an unaligned component of a packed or crammed structure, and when the by option is used, must be type integer or subrange thereof.

The sequence of values assigned to the control variable for which the statement list is executed is determined solely by the initial value, 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, 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 exit from the statement is a normal one, 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) ;
    control^ := temp ;
  whilend ;
  S;
ifend
end
```

10.0 STATEMENTS
10.2.6 FOR STATEMENTS

A for the 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
var  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 by 1 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 ; "note: by option
          is not allowed when the control variable is not
          of type integer or type subrange of integer."
```

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> ::= <simple constant expression>
                        [..<simple constant 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. If the value of the selector is not included in any selection spec and the else is omitted, the program is in error.

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

Selection specs are restricted to simple expressions (cf. 9.0) to preclude the use of unparenthesized relations as selection specs.

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

Examples:

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

10.0 STATEMENTS
0.2.7 CASE STATEMENTS

```
case i of
   =1=      x := sin(x) ;
   =2=      x := cos(x) ;
   =3=      x := exp(x) ;
   =4=      x := ln(x)
   else     x := -x
casend
```

" * * * * * * * "

```
type lextype = (basic, inconst, realconst, stringconst,
   identifier),

     symbol = record
                 case lex : lextype of
                 =basic= name : symbolid,
                         class : operation,
                 =inconst= value : integer,
                           optimiz : boolean,
                 =realconst= value : real,
                 =stringconst= length : 1..255,
                               stringbuf : ^string(*),
                 =identifier= identno : integer,
                              decl : ^symbolentry
                 casend
              recend

var cursym : symbol, sign : boolean := false ;

L1 : insymbol ;
L2 : case 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 ;
     =inconst= 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 (cursm.decl) else
       cursym := cursym.decl^.value^ ; goto L2
       ifend
     casend
```

10.2.8 VALUE CONFORMITY CASE STATEMENT

A value 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, and permits references to that value by assigning it to a variable. The union variable may be of packed union type.

<value conformity case statement> ::=
    case :=: <union variable> of <value conformity cases>
        [else <statement list>] casend

<union variable> ::= <variable>

<value conformity cases> ::=
    <a value conformity case> {; <a value conformity case> }

<a value conformity case> ::=
    =<value type specifier>= <statement list>

<value type specifier> ::= <variable>

Each value type specifier must be a variable of one (and only one) of the types of the union variable, and must be of a type different from all other value type specifiers in the statement (i.e., type selection must be unique). If one of the value type specifiers is of the type of value last assigned to the union variable, it will be assigned that value and the associated statement list will be executed. Within the statement list, the value type specifier (i.e., the variable itself) may be used to access the value which was that of the union variable.

If none of the value type specifiers matches the type of the value of the union variable, the statement list following else is executed. If none of the value type specifiers matches the type of the value of the union variable, and if the else part is omitted, the program is in error.

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

10.0 STATEMENTS
10.2.8 VALUE CONFORMITY CASE STATEMENT

Example:

```
var b2 : boolean, i2 : integer, r3, r4 : real,
    mix = union (integer, real, boolean) ;
          .
          .
          .  ;
    mix := (r3 + 24.213) / r5 ; "mix assumes a real value"
          .
          .
          .
    case :=: mix of "value conformity case"
      =b2=    stb := sta ; k:= k + 1 ;
      =i2=    i2 := i2 + 2 ; b2 := true ;
      =r4=    r3 := 2 * r4 ; b2 := true ;
                "r4 is assigned value of mix and
                 this case is chosen"
casend ;
```

10.2.9 POINTER CONFORMITY CASE STATEMENTS

A pointer 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, and permits references to that value by assigning a pointer-to-the-value to a pointer variable. The union variable may not be of packed union type.

<pointer conformity case statement> ::= case :^: <union variable>
of <pointer conformity cases>[else <statement list>] casend

<union variable> ::= <variable>

<pointer conformity cases> ::=
<a pointer conformity case>{;<a pointer conformity case>}

<a pointer conformity case> ::=
=<pointer type specifier>= <statement list>

<pointer type specifier> ::= <pointer variable>
<pointer variable> ::= <variable>

Each pointer type specifier must be a pointer variable to one (and only 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 pointer 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

10.0 STATEMENTS
10.2.9 POINTER CONFORMITY CASE STATEMENTS

the statement list, the pointer followed by an up arrow may be used to refer to the value.

If none of the pointer type specifiers matches the type of the value of the union variable, the statement list following else is executed. If none of the pointer type specifiers matches the type of the current value of the union variable, and if the else part is omitted, the program is in error.

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

Example:

```
proc format(ref u : union (integer,boolean), S : string(*)) ;
var pint : ^integer, pbool : ^boolean ;
case :^: u of  "pointer conformity case statement"
   =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. (Cf., chap. 8, Procs, Coprocs and Labels.) A procedure call statement may never be used to activate a function.

10.0 STATEMENTS
10.3.1 PROCEDURE CALL STATEMENT

```
<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.

10.3.1.1 Call_by_Value

A call by value parameter causes the establishment of a variable local to the called procedure, and the assignment of the value of the actual parameter to it. The type of the local variable is fixed as follows:

1. If the formal parameter is of fixed or variable bound type, then its instantaneous type is known at the time of call and becomes the type of the local variable. The actual parameter may be any expression which could be assigned to a variable of that type (cf. Assignment Statements).

2. If the formal parameter is of adaptable type, it must be an adaptable string, array, record, stack or sequence. The instantaneous type of the actual parameter must be one of those to which the adaptable type can adapt (cf. Adaptable Types), and the local variable takes on that type. The actual assignment of value then follows normal assignment rules.

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

4. If the formal parameter is a bound or 'unbound' variant record, then the actual may be an unbound or bound (respectively) variant record of the same instantaneous type.

10.3.1.2 Call_by_Reference

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 must be aligned (cf. Section 4.9) to ensure that it can be addressed.

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

1. If the formal parameter is of fixed or variable bound type, then its instantaneous type is known at the time of call. The actual parameter must be a variable or substring reference of the same instantaneous type, and that type is designated.

2. If the formal parameter is of adaptable type, the actual parameter must be a variable or substring reference whose instantaneous type is one of those to which the adaptable type can adapt (cf. Adaptable Types), and that type is designated.

3. If the formal parameter is a variant record then the actual parameter may not be a bound variant record.

4. 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.

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

10.3.2 CREATE STATEMENT

The create statement causes the creation of a coprocess from the specified procedure, and establisnes a new environment (including the actual parameter list, but not including automatic variables) 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. At that time, automatic variables are allocated.

10.0 STATEMENTS
10.3.2 CREATE STATEMENT

The procedure specified in the create statement is designated the primary procedure of the coprocess. An exit by means of a normal exit, return, or goto statement while the coprocess is still active is an error (cf. 8.2 for an integrated example and related semantics.)

It is possible to have several instances at the same time of a single procedure in use as a coproc, each instance having been created with a different pointer to coproc, and potentially with differing actual parameters. (See the second example following.)

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

<pointer to coproc> ::= <variable>

Example:

```
proc fixer(ref a, b, c : string (20) of char,
           answer : 1 .. 10) ;
             ...
procend
             ...
var  fixk1, fixk2, fixk3, fixk4, fixk4 : ^coproc ;
       "pointer to coproc"
             ...
create(fixk1,fixer(va, vb, str#rr, response#1)) ;
create(fixk2, fixer(va, vb, var#mm, str#ss, response#2)) ;
             ...
create(fixk5, fixer(rk,st,k24, str#ab, response#5)) ;
```

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 (cf. 8.2 for an integrated example and related semantics).

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

Example:

10.0 STATEMENTS
10.3.3 DESTROY STATEMENT

destroy (fixk2)

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 (cf. 8.2 for an integrated example and semantics).

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>^

Example:

resume (fixk2^)

10.3.5 CYCLE STATEMENT

The cycle statement allows the conditional by-passing of the remainder of the statement 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 cycle 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)

10.0 STATEMENTS
10.3.5 CYCLE STATEMENT

re-executing the statement list of a repetitive statement such as for, repeat, loop, or while.

Examples:

```
for i := 1 to n do
  cycle when x < a[i] ;
  x := a[i] "balance of loop skipped when x < a[i]"
forend
```

10.3.6 EXIT STATEMENT

The exit statement causes execution to continue at the successor of a designated structured statement or procedure when a condition is true or non-existent.

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

<label or proc identifier> ::= <label>|<procedure identifier>

If no label or procedure is specified, then execution continues at the successor of the immediately containing structured statement or procedure.

If a procedure is designated as the object of the exit, then that procedure must statically encompass the exit statement within the same compilation unit (see section 8.1.3 for exits from functions). If a label is designated as the object of the exit, then that label must be for a structured statement which statically encompasses the exit statement within the same compilation unit.

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

Examples:

```
repeat exit when key = a[i] ; i := i + 1 until i > n
        ...
L2:     x := y + 27 ; "example of exit <label>"
L3:     for k := 1 to 10000 do
          j := k ;
          if (i mod 2) = 0 then
            b[k] := false
```

```
          else
            prime(i, answer) ; "test if prime"
            loop
              if fibonacci(answer) then
                exit L3 "goes to L4"
              ifend ;
              answer := answer - 5 ;
              if answer <= 0 then
                exit L3 "exit: if, loop, if, for"
              ifend
            loopend
          ifend
        forend ;
L4:     "exit causes control to transfer here"
        bound := j ; ...
```

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 labeled statement designated 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

10.0 STATEMENTS
10.3.8 GOTO STATEMENT

than the one in which the goto statement occurs, then the goto statement is in error.

Examples:

```
proc     finder (ref a,b : real ) ;
         "example of goto with and without exit option"
label     L1, L2, L3 ;

var       k1, k2 : real, kint, jint : integer ;
proc      circle(ref k2 : real, val k1 : real) ;
L1:      k2 := $real(kint) ;
L3:      k1 := pi * k2 ;
         if k1 < $real(jint) then
" * "      goto exit L2 "go to L2 outside of procedure circle"
         ifend ;
         k2 := k2 + 0.05 ;
" * "    goto L3 ; "stays inside procedure circle "
procend circle ;

L1:      k1 := 2 * a + b ; ...  ; a := a + ...  ;
         circle(a, k2) ;
         goto L1 ;
L2:      if ...  ; "goto exit from procedure circle
                    comes to this point"
         k1 := ...  ; ...  ifend
L3:      if k1 ...  ifend
procend finder ;
```

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 is a structure into which other variables may be placed, referenced, and deleted under program control according to the discipline implied by the type of the storage variable. Storage management statements are the means for effecting this control.

10.0 STATEMENTS
10.4 STORAGE MANAGEMENT STATEMENTS

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

Prior to the first allocation into a storage variable, a reset statement is required for that storage variable.

10.4.1 ALLOCATION DESIGNATOR

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

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

or

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

```
<allocation designator> ::=
      <pointer variable>
     |<adaptable pointer variable> : [<adaptable field fixer>
       {,<adaptable field fixer>}]
     |<pointer to bound variant record variable> :
       [<tag field fixers>]

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

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

<adaptable field fixer> ::= <star fixer>
                           |<starry subrange fixer>
                           |<length fixer>
                           |<span fixer>

<star fixer> ::= <scalar expression> .. <scalar expression>
<starry subrange fixer> ::= <scalar expression>
```

10.0 STATEMENTS
10.4.1 ALLOCATION DESIGNATOR

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

1. Star fixers and starry subrange fixers are used in the fixing of adaptable bounds of arrays.

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

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

4. The order, types, and values of adaptable field fixers must select one of the types to which the associated adaptable pointer can adapt (cf. Adaptable Types).

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

6. For the bounds list used in an allocation designator, entries are reqired only for dimensions (or either end of a dimension) which are adaptable. No place holders or position markers are required to indicate values or relative positions within the entire sequence of the fixed or variable bound indices involved in the total type definition.

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

Example:

```
type tipe = array [1..5, * , v2..v3, 21..*] of real ;
var point : ^tipe , bunch : heap (rep 25000 of real) ;
        "point is an adaptable pointer variable"
          ...
allocate point : [5..15,24] in bunch ;
```

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

1 to 5, 5 to 15, v2 to v3, and 21 to 24.

and would set the pointer variable, point, to designate that array. (The values of v2 and v3 will have been established on entry to the block containing the above declarations.)

A subsequent statement of the form:

allocate point : [r..s, t] in bunch ;

would allocate an array with dimensions

1 to 5, r to s, v2 to v3, and 21 to t

and would reset the pointer variable, point, to designate the new array. The second and fourth dimensions will be determined by the values of r,s, and t when the statement is executed.

10.4.2 PUSH STATEMENT

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

The first push statement for any user-declared stack must be preceeded by a reset statement for the stack, or the program will be in error. (Such a reset statement is neither required nor possible by the user for the 'System Stack').

<push statement> ::= push <pointer variable> on <stack variable>
| push <allocation designator>

<stack variable> ::= <variable>

10.4.2.1 User-Declared_Stack

The object type (the type pointed-to) of the pointer variable must be the same as the stack's component type. Space for a variable of that type is allocated atop the specified stack, and the pointer variable is set to designate the newly allocated variable (or to the value nil if there is insufficient space). The pointer variable may then be used to access the newly allocated variable.

10.0 STATEMENTS
10.4.2.1 User-Declared Stack

Example:

```
var stk : stack [100] of integer,
     stktop : ^integer, "to point to top of stack 'stk' "
          k : integer := 1 ;
     push stktop on stk ; "allocate space for new value"
     stktop^ := k ; "assign new value"
```

10.4.2.2 System-Managed Stack

If a stack is not specified, space for a variable of the type determined by the allocation designator is set to designate that variable (or to the value nil if there is insufficient space). A variable so allocated can not be explicitly de-allocated by the user. Instead, de-allocation occurs automatically on exit from the block (cf. 7.5) containing the allocating push statement, at which time space for the variable is released and its value becomes undefined.

Example:

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

     "allocate space for array [20] of integer on

     system stack, i-th element can be referenced

     as localarray^[i] "
```

10.4.3 POP STATEMENT

The pop statement causes the de-allocation of the top element on a specified stack (space for the variable is released and its value becomes undefined), and a pointer variable is set to designate the previous variable on the stack (which becomes the new top of stack). If no elements remain on the stack, the pointer variable is set to the value nil.

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

The object type of the pointer variable (the type pointed-to) must be the same as the stack's component type, and the use of a read-only pointer variable is an error.

10.0 STATEMENTS
10.4.3 POP STATEMENT

Examples:

```
var   intstack : stack [100] of integer,
      intstacktop : ^integer,
      j, k : integer ;
    reset intstack; ...

      for k := 1 to 90 do
        push intstacktop on intstack ;
        intstacktop^ := 2 * k ; "2, 4, 6, ...  , 180"
      forend ;

      for k := 1 to 15 do
        pop intstacktop on intstack;
      forend ;

      "seventy-five values now in stack"
      j := intstacktop^ ; "j has value 150 "
```

10.4.4 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.

Prior to the first next statement for any sequence, the program must execute a reset for that sequence to set it to the beginning, or the program will be in error.

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

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

<sequence variable> ::= <variable>
```

Example:

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

10.4.5 RESET STATEMENT

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

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

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

10.4.5.1 Reset_Sequence

The reset sequence statement causes positioning in a sequence. The current element of the sequence becomes either the first element or the element specified by the pointer variable. The use of a pointer variable whose value had not been set by a next statment for the same sequence, or whose value is nil, is an error.

Example:

```
reset buf to length_ptr
```

10.4.5.2 Reset_Stack

The reset stack statement causes de-allocation and positioning in a stack. If a pointer variable is not specified, all elements of the stack are de-allocated. If a pointer variable is specified, its object type (the type pointed-to) must be the same as the stack's component type, and the operation is as follows: if the pointer variable does not designate the top of the specified stack, then the top element is de-allocated and the process continues until the pointer variable designates the top element. The use of a pointer variable whose value had not been set by a push or pop operation on the specified stack, or whose value is nil, is an error.

10.4.5.3 Reset_Heap

The reset heap statement causes all elements currently allocated (cf. 10.4.6) in the specified heap to be freed (cf. 10.4.7) en-masse.

10.4.6 ALLOCATE STATEMENT

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

Note that the first allocate statement for any heap must be preceeded by the execution of a reset statement for that heap, or the program will be in error (cf. 10.4.5).

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

<heap variable> ::= <variable>

Examples:

```
var my_stack : ^stack [*] of integer;

allocate my_stack : [50]; "allocate space in system heap"
allocate sym_ptr in symbol_table
```

10.4.7 FREE STATEMENT

The free statement causes the deletion of a specified variable from a heap, making its storage available for subsequent allocate statements.

A pointer variable specifies the variable to be freed. If the pointer variable was not set as a result of a previous allocate statement for the same heap, the effect is undefined. Execution of the free statement sets the pointer variable to the value nil. Use of a pointer variable with a value of nil to attempt data access is an error.

10.0 STATEMENTS
10.4.7 FREE STATEMENT

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

Examples:

```
free sym_ptr in symbol_table
free my_stack
```

10.5 INPUT-OUTPUT STATEMENTS

Four file types are accommodated:

Legible_files consist of a sequence of entities called lines. System-defined mappings between lines and strings_of_char exist; these may differ depending on the source or destination device of the lines.

Print_files are special cases of legible files that permit the user to control output formatting through the use of pagination, spacing and titling procedures (permitted on print files only), rather than through the use of embedded control characters. The user should not directly embed such control characters in data destined for print files.

Binary_files consist of a linear sequence of SWL variables. These variables are not self-identifying, so that results of a read operation are guaranteed if and only if the sequence of types read is the same as the sequence written.

Direct_files are special cases of binary files that also permit the retrieval (and rewriting) of variables 'directly' through the use of a 'key'. Results of such a read (or rewrite) operation are guaranteed if and only if the obvious (but tediously described) type matching holds.

Files are accessed through file_variables (cf. 6.7), which are associated with a file by an explicit open procedure and de-associated from a file by an explicit close procedure. File variables take on as values an undefined record structure whose component values specify the kind and current state of the associated file. A file spec (cf. 6.7.1) is used to specify 'initial' values consisting of the actual name of the file and certain other file attributes.

Warning: A file-variable may be assigned to another file-variable of the same type (and only the same type). If this occurs, and the program attempts use of both file-variables to manipulate the same file in an interchangable (intermixed)

10.0 STATEMENTS
10.5 INPUT-OUTPUT STATEMENTS

manner, operations done via one of the file variables will not be reflected in the other, and serious errors can occur. (Note that the passing of a file variable by value to a procedure can create analogous errors if dual file variable usage is attempted for a single file.)

Input-output statements are used to associate and de-associate file variables and actual files, to position files, to transmit information to and from files, and to format print files.

```
<input output statements> ::= <open statement>
                            | <close statement>
                            | <positioning statements>
                            | <read-write statements>
                            | <format control statements>
```

10.5.1 OPEN STATEMENT

An actual file is inaccessible until it has been associated with a file variable. This is accomplished through an open statement.

<open statement> ::= #open (<file variable> [, <file spec>])

The open statement can specify or respecify file attributes (including the actual file name) associated with the file variable to be used for accessing the file; however, the file type can not be respecified.

It is an error to open a file which is already opened.

10.5.1.1 Unspecified_Attributes

To be provided.

10.5.2 CLOSE STATEMENT

The close statement prevents further access to the actual file until a subsequent open of that file is executed.

<close statement> ::= #close(<file variable>)

10.5.3 POSITIONING STATEMENTS

Positioning statements permit an actual file to be positioned at its beginning (through a #first statement), at its end (through a #last statement), or at some position specified by a key associated with a direct file (through a reset statement).

```
<positioning statements> ::=
                  #first (<file variable>)
                | #last (<file variable>)
                | #reset(<direct file variable>,<key value>)

<direct file variable> ::= <file variable>

<key value> ::= <integer expression>
```

10.5.4 READ-WRITE STATEMENTS

```
<read-write statements> ::=
                      <write line statement>
                     |<write partial line statement>
                     |<write binary statement>
                     |<write sequential statement>
                     |<write direct statement>

                     |<read legible statement>
                     |<read partial legible statement>
                     |<read binary statement>
                     |<read sequential statement>
                     |<read direct statement>
```

10.5.4.1 Write_(Partial)_Line_Statement

These statements cause the conversion of value(s) into string form and their transmission as a line or as part of a line.

10.5.4.1.1 WRITE LINE STATEMENT

The write line statement causes the conversion of a value or a sequence of values into string form, according to specified formats for each of the values, and the transmission of those concatenated strings to the specified legible or print file as a line. The formatting information is contained within the write line statement itself, or is the system-default standard (cf.

10.0 STATEMENTS
10.5.4.1.1 WRITE LINE STATEMENT

10.5.4.2).

<write line statement> ::= #put (<file variable>,
<put element> {,<put element>})

10.5.4.1.2 WRITE PARTIAL LINE STATEMENT

The write partial line statement causes the conversion of a value or a sequence of values into string form, according to specified formats for each of the values, and the transmission of those concatenated strings as a part of a line to the specified legible or print file. The formatting information is contained within the write partial line statement itself, or is pre-defined (cf. 10.5.4.2).

Successive write partial line statements will concatenate (left to right) the strings from each succesive write partial line statement into the same line of the specified legible or print file until the boolean expression which indicates the 'final part of the line' is true. The write partial line statement execution for which the boolean expression is true will cause both the transmission of the values (put elements) of that statement and the line to be 'completed'.

The program will be in error if a #putpart for a file with the <boolean expression> value false is followed in execution (for the same file) by a #put, #get, or a #getpart instead of a further #putpart. The final #putpart for the record must have a <boolean expression> value of true or the program is in error.

<write partial line statement> ::= #putpart (<file variable> ,
<boolean expression>,<put element> { ,<put element>})

10.5.4.2 Put Elements

Values to be output by write line and write partial line statements, rules for their conversion into strings of characters, and the sizes and formats of the receiving fields for such strings are specified by put elements.

<put element> ::= <scalar element>
| <string element>
| <real element>
| <pointer element>

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

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

10.0 STATEMENTS
10.5.4.2 Put Elements

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

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

<radix expression> ::= <"an expression whose value
is a valid radix, cf. 3.2.4">

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

<string field specifier> ::=
<empty>
| :<radix spec>
| :<field length> [:<radix spec>]
| :<field length> :<positions/char> [:<radix spec>]
| :*:<positions/char> [:<radix spec>]

<positions/char> ::= <positive integer expression>

<real element> ::= <real expression> [<real specifier>]

<real specifier ::= :<radix spec> | <empty>

| :<field length> [:<right of point>]

<right of point> ::= <non-negative integer expression>

In general, values specified to be in numeric form are written right justified into the specified field, with blank left fill or truncation on the left. Values specified to be in string or character (alphabetic) form are written left justified into the specified field, with blank right fill or truncation on the right.

10.5.4.2.1 CHARACTER CLASS OUTPUT

If the put element is :

(a) a string without radix spec

(b) a character without radix spec

(c) a boolean without radix spec

(d) an ordinal without radix spec

Then

(1) The string value of the expression is placed left

justified into the field length specified. For strings and characters, the character from the ASCII character set is supplied. For booleans the string 'true' or 'false' is used. For ordinals a string identical to the ordinal constant identifier corresponding to the ordinal value is used.

(2) If no field length is specified, then a field is furnished equal in length to the value of the string.

(3) If the length of the string value is greater or less than the specified field length, then right truncation or right blank fill respectively will occur.

10.5.4.2.2 NON-NUMERIC SCALAR WITH RADIX

If the put element is:

(a) a character expression

(b) a boolean expression

(c) an ordinal expression

then:

(1) The $integer function is applied to the value of the expression to obtain an integer.

(2) The value of the integer is expressed in the radix specified , is right-justified in the field length specified, with blank fill to the left.

(3) If no field length is specified, then for the put elements and radices as listed below, the field length default values are:

element	radix	length
character	decimal	4
character	octal	4
character	hexadecimal	3
boolean	decimal	1
boolean	octal	1
boolean	hexadecimal	1
ordinal	decimal	(as for integer elements)
ordinal	octal	(as for integer elements)
ordinal	hexadecimal	(as for integer elements)

(4) If the field length is specified but is shorter than the

representation of the value then left truncation can occur.

10.5.4.2.3 STRING ELEMENT WITH RADIX

Each character of the string is individually converted to an integer by the $integer function, and then placed right justified into a field of size positions/char. If positions/char is omitted, then the default sizes for each radix are the same as for a character put-element (decimal - 4, octal - 4, hexadecimal - 3). These representations are placed left justified into the specified field length, or if no field length is furnished, then a field is supplied equal in length to positions/char times the number of characters in the string. Truncation or blank fill on the right may occur, when the field length is specified. (Truncation or blank fill may also occur at the character representation level when positions/char is specified with a radix spec also present.)

10.5.4.2.4 REAL ELEMENT

If only a radix spec is given as the real specifier, than an implementation defined form of the internal representation will be produced in the specified radix.

If only a field length is given as the real specifier, or no specifier is present, then the value of the real expression is converted to a standard decimal floating point form, right justified in a field of the length specified, with blank left fill or left truncation if the length specified is longer or shorter than the required space for the standard representation.

If the real specifier is of the form <field length><right of point>, then the real expression value is converted to a decimal representation of its value which has <right of point> digits to the right of the decimal point and an explicit decimal point between the integer and non-integer parts of the number. If the value is negative, then a minus sign will be placed to the left of the most significant digit to the left of the decimal point.

Any excess positions on the left will be blank filled. If there are no significant digits to the left of the decimal point, then one zero will be placed to the left of the point, with left fill of blanks (and optionally the minus sign preceding the zero, when the value is negative).

Field overflow can cause loss of the minus sign (if present), of high order digits, and of the decimal point, by truncation on the left.

10.0 STATEMENTS
10.5.4.2.5 INTEGER ELEMENT

10.5.4.2.5 INTEGER ELEMENT

If a scalar field specifier is given which is only a field length, then the value of the integer expression is converted to a standard decimal radix notation and placed right justified into the field, with blank fill to the left. If the field length given is not long enough to contain all the digits of the value of the integer expression, then left truncation of the high order integer digits occurs. If the integer expression is negative in value, then a minus sign precedes the leftmost significant digit within the field (but might be a part of left truncation if the value is too large).

If the scalar field specifier is omitted (empty), then the same process is performed as described in the preceding paragraph, but the value is placed right justified into a field of 'standard length' (which is machine and implementation dependent) with blank fill to the left.

If a scalar field specifier is furnished which contains a radix spec, then the integer expression value is converted to a standard notation in the radix indicated, and placed right justified into the field of specified length. If the field length is omitted, then placement is into a 'standard length' field -- which is both machine and implementation dependent and radix dependent, with blank fill to the left. Again, left truncation can occur.

10.5.4.2.6 SCALAR SUBRANGE ELEMENT

A put element which is a scalar subrange type is handled exactly as the scalar range of which it is a subrange.

10.5.4.2.7 POINTER ELEMENT

A pointer element may be either a pointer or a relative pointer. If a radix spec is included in the pointer element, then an implementation dependent form of the pointer will be written in the specified radix system.

If the radix spec is omitted, then the implementation dependent form of the pointer will be written in the radix 10 system. The form written will contain a system-standard indication if the pointer is a relative pointer (or will have a differing standard format than a non-relative pointer).

In either case, the field size for pointer representations will be standard (but will be system-dependent).

10.5.4.3 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)
```

10.5.4.4 Write_Sequential_Statement

A write sequential statement causes the value of an expression to be transmitted to the specified direct file. It also assigns a value to the key variable specifying the position on the file of the expression written.

```
<write sequential statement> ::= #put (<file variable> ,
                                 <key variable>,<expression>)

<key variable> ::= <integer variable>
```

Example:

```
#put(symbol_file, symbol_key, symbol_value )
```

10.5.4.5 Write_Direct_Statement

A write direct statement causes the value of an expression to be transmitted to the specified direct file at the location specified by the key value. The write direct statement does not modify the current file position.

```
<write direct statement> ::= #putdir (<file variable> ,
                                 <key value> ,<expression>)

<key value> ::= <integer expression>
```

Example

```
#putdir (symbol_file, key , symbol_table_entry )
```

10.5.4.6 Read_Legible_Statement

The read legible statement, #get, causes that portion of the next line which will fit left justified in the specified string variable to be moved. If the line is less than or equal to the length of the string variable, the boolean value true will be returned in the boolean variable. If the line is greater than the length of the <string variable> part of the line is returned in the <string variable> and the boolean value false is returned in the <boolean variable>. In both cases, the number of characters returned is returned in <no_read>. If only a partial line was returned, the remainder may be obtained through subsequent #getpart requests.

An attempt to read beyond the last line causes the boolean variable to be set to true, <no_read> to be set to zero, and the built-in function #eof (<file variable>) to return the boolean value true. The string variable is unmodified.

If a #getpart statement which returns false or true into the <boolean variable> is sequentially followed by a #get statement, then any balance of the current record is bypassed and the #get statement returns the values from the next record of the file.

```
<read legible statement> ::=
     #get(<file variable>,<boolean variable>,<no_read> ,
                                          <string variable> )

     <string variable> ::= <variable>
     <no_read> ::= <integer variable>
```

Example

```
     #get (source_file, lastline,line_length, line_buffer)
```

10.5.4.7 Read_Partial_Legible_Statement

The read partial legible statement causes that portion of the next line which will fit left justified in the specified string variable to be moved. If the line is less than or equal to the length of the string variable, the boolean value true will be returned in the boolean variable. If the line is greater than the length of the <string variable> part of the line is returned in the <string variable> and the boolean value false is returned in the <boolean variable>. In both cases, the number of characters returned is returned in <no_read>. If only a partial line was returned, the remainder may be obtained through subsequent #getpart requests.

10.0 STATEMENTS
10.5.4.7 Read Partial Legible Statement

An attempt to read beyond the last line causes the boolean variable to be set to true, <no_read> to be set to zero, and the built-in function #eof (<file variable>) to return the boolean value true. The string variable is unmodified.

The program is in error if a #getpart statement which returns false into the <boolean variable> is sequentially followed by a #put or a #putpart. If the #getpart statement which returns false into the <boolean variable> is sequentially followed by a #get statement, then the balance of the current record is bypassed and the #get statement returns the values from the next record of the file.

<read partial legible statement> ::=
#getpart(<file variable>,<boolean variable>,<no_read> ,
<string variable>)

Example

```
repeat
#getpart (source_file, end_line, line_length, line_buffer);
accum_string (line_buffer, line_length);
until end_line;
```

10.5.4.8 Read_Binary_Statement

The read binary statement causes the transmission of a value from a binary 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, and the variable is unmodified.

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

Example:

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

10.5.4.9 Read_Sequential_Statement

The read sequential statement causes the transmission of a value from a direct file to a variable. It also assigns a value to the key variable specifying a position on the file of the value obtained. An attempt to read beyond the end of information causes the built-in function #eof(<file variable>) to return true , and the value of the variable is unchanged.

10.0 STATEMENTS
10.5.4.9 Read Sequential Statement

<read sequential statement> ::= #get (<file variable>,
<key variable> , <variable>)

Example:

#get(symbol_file, symbol_key, symbol_string)

10.5.4.10 Read_Direct_Statement

The read direct statement causes the transmission of the value associated with the key from the file to a variable. The type of the variable must be the same as the type of the value in the write direct statement that associated the value with the key. The current file position is unmodified by this statement.

<read direct statement> ::=
#getdir (<file variable> , <key value> , <variable>)

Example:

#getdir (symbol_file, symbol_key, symbol)

10.5.5 FORMAT CONTROL

Print files may be formatted into pages consisting of a specified number of lines. The number of lines per page (pagesize) can be set or reset by the user. Lines are numbered from one (1) to #pagesize. The number of the next line to be printed is called the current line number. The user can interrogate the current line number via the #curline function, and can use format control statements to exercise direct control over pagination and lineation.

In addition, whenever the number of lines actually printed would exceed #pagesize, an end-of-page regimen is invoked. The end-of-page regimen consists of either an eject operation (see below) or the invocation of a user specified end-of-page procedure (cf. 6.7.1.1).

10.0 STATEMENTS
10.5.5 FORMAT CONTROL

<format control statement> ::= <page statement>
| <eject statement>
| <line statement>
| <skip statement>

10.5.5.1 Page Statement

The page statement invokes the end-of-page procedure (cf. 6.7.1.1) directly; if none has been specified for the file, the #eject procedure is used(see below).

<page statement> ::= #page (<print file variable>)

10.5.5.2 Eject Statement

The eject statement issues a conventional page eject and sets the current line number of the specified file to one (1).

<eject statement> ::= #eject (<print file variable>)

10.5.5.3 Line Statement

The line statement spaces a print file forward to a target line number on the same page if the target line number lies between the current line number and the page size. If the target line number exceeds the page size, then the current line number is set to (pagesize+1) and the end-of-page regimen is invoked.

If the target line number is less than or equal to the current line number, the current line number is set to (pagesize+1), the end-of-page regimen is invoked, and the current line number is tested again: if the target line number lies between the current line number and the page size, then the lineation is carried out; otherwise, no further action is taken.

<line statement> ::= #line(<print file variable>,<line number>)

<line number> ::= <positive integer expression>

The related function #curline (cf. 11.2.20) may be used to find the current line number for a print file.

10.5.5.4 Skip_Statement

The skip statement spaces a print file forward either one line or the specified number of lines. If the new line number would exceed the page size, the current line number is set to (pagesize+1) and the end-of-page regimen is invoked.

<skip statement> ::=
    #skip (<print file variable> [,<number of lines>])

<number of lines> ::= <positive integer expression>

1.0 STANDARD PROCEDURES AND FUNCTIONS

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

11.1.1 #TRANSLATE (S, D, T)

This procedure accepts as arguments the string variables S, D, and T, and assigns to D the result of converting the elements of S according to the 'translation table,' T, by the following algorithm:

```
if ((1 <= #curstrlength(s) ) and
       (#curstrlength(s) <= #strlength(D) )) then
  for i := 1 to #curstrlength(s) do
    D(i) := T( 1 + $integer(S(i)))
  forend
else
  "error"
ifend
```

11.1.2 #STRINGREP (V, D, W[,R])

This procedure accepts as arguments: An expression, V, of type integer, real, or boolean; a string variable, D; an integer expression, W; and an optional integer expression, R.

The string D takes on the string representation of the value of V, in the following manner:

1. Boolean values are converted to one of the strings "true" or "false".

2. Integer values are converted to W-digit decimal numerals

11.0 STANDARD PROCEDURES AND FUNCTIONS
11.1.2 #STRINGREP (V, D, W[,R])

with leading zeros replaced by blanks, and a minus sign preceding the most significant digit for negative values.

3. Real values are converted into a string representation which assumes the radix 10, as defined for put elements in 10.5.4.2.4. The parameters to the procedure are:

V is the real number to be converted

D is the string reference for destination of the value

W is the field width

R is the optional <right of point> specification.

Blank fill and truncation rules are the same as those given in 10.5.4.2.4 for real put elements.

4. The string resulting from these operations is left extended, if necessary, by blanks to match the length of D, and then assigned to D.

11.1.3 #SETPAGESIZE (<PRINT FILE VARIABLE>,<NUMBER OF LINES>)

This procedure sets the page size for the specified print file variable to the value of the integer variable, number of lines.

11.1.4 #SETPAGEPROC (<PRINT FILE VARIABLE>,<PROCEDURE REFERENCE>)

This procedure sets the end-of-page procedure for the specified print file variable to be the procedure specified. This procedure can be invoked by a page statement (cf. 10.5.5.2), and will be automatically invoked whenever the current line number for the print file exceeds that file's specified page size. See section 6.7.1.1 for end-of-page procedure conventions.

11.2 STANDARD FUNCTIONS

The following standard functions return values of the specified type.

11.0 STANDARD PROCEDURES AND FUNCTIONS

11.2.1 #ABS(X)

Computes the absolute value of x. The type of the expression, 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.

The type of the expression, x, must be integer or real, and the result is the same type as x.

11.2.3 #SUCC(X)

The type of the expression, x, must be scalar or subrange, and the result is the successor value of x (if it exists).

11.2.4 #PRED(X)

The type of the expression, x, must be scalar or subrange, and the result is the predecessor value of x (if it exists).

11.2.5 $INTEGER(X)

Returns the integer representation of the value x. The type of the expression, x, must be ordinal, char, boolean, or real. If x is real then the value returned is an integer y of the same sign as x, such that

$$abs(x)-1 < abs(y) \leq abs(x).$$

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

11.2.6 $REAL(X)

Returns a value of type real that approximates the value of the integer expression, x. Note that $integer($real(x)) does not necessarily equal x.

11.2.7 $CHAR(X)

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

11.2.8 $STRING(L,S[,FILL])

L is an integer expression, s is a string expression, and fill is a character expression.

Returns a string value of length l from the string or substring 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 of the string x. For a fixed string this is the allocated length, and x may be either a string variable or a string type identifier. For a varying string this is the maxlength and x may be either a string variable or a string type.

11.2.10 #LOWERBOUND(ARRAY, N)

Returns the value of the n-th lower bound of the array, where the leftmost subscript position is numbered 1. The type of the result is the index type of that dimension of the array. The argument (array) may be either a array variable or an array type identifier. N must be a <constant integer expression>.

11.2.11 #UPPERBOUND(ARRAY, N)

Returns the value of the n-th upper bound of the array, where the leftmost subscript position is numbered 1. The type of the result is the index type of that dimension of the array. The argument (array) may be either an array variable or an array type identifier. N must be a <constant integer expression>.

11.2.12 #EOF(FILE)

Returns the value true if the end-of-file condition exists for the specified file. Returns false otherwise. The argument must be a file variable.

The current SWL end-of-file remains operating system independent, and is defined by the last sequential write of a legible, binary, or direct file. Thus, the close statement for a file being written in a sequential manner will implicitly also write an end-of-file indication (which will be system dependent in form, but detected by #eof function on a system-independent basis).

Note that a write direct statement (#putdir) is essentially used to modify and update an existing file. Thus, writes done with #putdir do not change the location which will be 'end-of-file' as caused by writing in sequential manner and then closing tne file.

11.2.13 #COPROCID

Returns the value of type pointer to coproc of the coprocess in which it is executed. (See 8.2 for an example in context.)

11.2.14 #REL(POINTER[,PARENTAL])

Produces a relative pointer value from a pointer variable and parental variable. If the parental variable is not supplied, the default heap is used. The relative pointer's object type is the object type of the pointer variable, and its parental type is that of the parental variable. The result is undefined if the pointer does not designate an element of the parental variable. (See also example under 11.2.15.)

11.2.15 #PTR(RELATIVE_POINTER[,PARENTAL])

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

Example:

```
type myheap = heap (rep 500 of integer) ;
var rptr1, rptr2, rptr3 : rel (myheap) ^integer,
    h : myheap,
    ipt : ^integer,
    a, b, x : integer,
    cptr, dptr : (myheap) ^integer ;
       ... ; reset h ;
x := a + 3 ; ...
b := x + 2 * a - 7 ;
allocate dptr in h ; "get space in heap h"
dptr^ := b ; "put value in allocated variable"
allocate cptr in h ; "get another space in heap h"
cptr^ := x ; "put value in allocated variable"
rptr1 := #rel(dptr, h) ;
rptr2 := #rel(cptr, h) ; "record current relative position of
                           cptr in h"
       ...
allocate cptr in h ; "get new variable in heap"
       ... ;
free cptr in h ; "free space in heap"
cptr := #ptr(rptr2, h) ; "point cptr back to another
                          remembered variable in heap h"
```

11.2.16 #UPPERVALUE(X)

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

11.0 STANDARD PROCEDURES AND FUNCTIONS
11.2.17 #LOWERVALUE(X)

11.2.17 #LOWERVALUE(X)

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

11.2.18 #PREVIOUS(S,N)

Accepts as arguments a stack variable, S, and an expression, N, which must yield a non-negative integer value. The function returns a pointer value which designates that element of the specified stack which would be the topmost element were N pops to be executed on the specified stack.

11.2.19 #CURPAGESIZE (<PRINT FILE VARIABLE>)

Returns the print file's page size as an integer.

11.2.20 #CURLINO (<PRINT FILE VARIABLE>)

Returns the print file's current line number as an integer.

11.2.21 #CURSTRLENGTH (X)

Returns the current length (as an integer) of a string, which is defined as follows: the current length of a varying string is defined to be N whenever its value is a fixed string of length N and is defined to be zero whenever its value is the null string; the current length of a fixed string is equal to its length.

11.2.22 $BOOLEAN (X)

Returns the value false if x mod 2 = 0 (where x is always a non-negative integer value). If x mod 2 = 1 then the function returns the value true.

11.0 STANDARD PROCEDURES AND FUNCTIONS
11.3 REPRESENTATION DEPENDENT

11.3 REPRESENTATION DEPENDENT

11.3.1 #LOC(<VARIABLE>)

Returns a pointer to the argument which can be directly assigned or compared to any direct pointer type (but not to a relative pointer type).

11.3.2 #SIZE(ARGUMENT)

Returns the number of cells (cf. Section 13.1.1, Cell Type) required to contain a variable of the same type as the argument. The argument may be either a variable or a type identifier.

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. $0 \leq n < base$.

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. The first argument may be either a variable or a type identifier.

11.4 SYSTEM DEPENDENT FUNCTIONS AND PROCEDURES

Since many aspects of I/O are system dependent, the language includes facilities for checking and creating system-dependent file-structure delimiters. The spelling of such facilities indicates that dependence, and the identifiers for Cyber-related facilities are distinguished by the prefix '#6'. The Cyber related facilities are the creation procedures #6mkeor and #mkeof, and the checking functions #6eor, #6eof, and #6eoi.

11.4.1 #6WEOR (<FILE VARIABLE>)

The #6weor standard procedure writes an 'end-of-record' as denoted in the CYBER system. The indication of reaching that indicator when reading a file will be detected by use of the #6eor function (cf. 11.4.3).

11.4.2 #6WEOF (<FILE VARIABLE>)

The #6weof standard procedure writes an 'end-of-file' as denoted in the CYBER system. The indication of reaching that indicator when reading a file will be detected by use of the #6eor function (cf. 11.4.4).

11.4.3 #6EOR (<FILE VARIABLE>)

The #6eor standard function returns the boolean value true if the file (in CYBER) is at an 'end-of-record' as a result of the last read command which contained valid data. (i.e., the test for an 'end-of-record' condition should be made prior to performing a read for the further data of the record.) Otherwise, the function will return the value false.

11.4.4 #6EOF (<FILE VARIABLE>)

The #6eof standard function returns the boolean value true if the file (in CYBER) is at an 'end-of-file' as a result of the last read command which returned valid data of the file; otherwise it returns the value false. (i.e., the test for an 'end-of-file' condition should be made prior to performing a read for further data of the file.)

11.4.5 #6EOI (<FILE VARIABLE>)

The #6eoi standard function returns the boolean value true (in CYBER) if the last read command giving valid data also returned an 'end-of-information' indicator; otherwise it returns the value false.

12.0 COMPILE-TIME FACILITIES

12.0 COMPILE-TIME FACILITIES

Compile-time facilities are essentially extra-linguistic in nature in that they are used to construct the program to be compiled rather than having a meaning in the program itself.

The facilities consist of the compile-time variable declarations, compile-time statements, and macro facilities. They generally apply from the point of definition until the end of the compilation unit, or until a compile-time redefinition of the same identifier. The result of processing the text containing these facilities must be a properly block-structured compilation unit.

12.1 STATEMENTS AND DECLARATIONS

12.1.1 COMPILE-TIME VARIABLES

Compile-time variables of type integer and boolean may be declared by means of the compile-time declaration statement.

```
<compile-time declaration> ::= ? var <compile-time var spec>
   {,<compile-time-var spec>} ?;
<compile-time var spec> ::=
   <identifier list> : <compile-time type> := <expression>
<compile-time type> ::= integer ! boolean
```

The following rules apply:

1. The compile-time declaration statement must appear before the use of any of the compile-time variables. The scope of the compile-time variable is from the point of declaration to the end of the compilation unit.

2. Compile-time variables may be used within compile-time expressions and compile-time assignment statements.

3. The expression used to initialize a compile-time variable must be composed only of constants and compile-time variables, but excluding identifiers for user-defined constants.

12.0 COMPILE-TIME FACILITIES
12.1.1 COMPILE-TIME VARIABLES

4. The operators defined on compile-time variables are:

```
+ - * / mod        for type integer
and or xor not     for type boolean
< = > <= >= /=     for type integer and boolean
```

5. Identifiers of compile-time variables may not be the same as any other program identifiers, including macro identifiers.

12.1.2 COMPILE-TIME ASSIGNMENT STATEMENT

The value of a compile-time variable may be altered by a compile-time assignment statement.

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

The following rules apply:

1. The variable must be a compile-time variable, and the expression must be composed only of constants and compile-time variables, but excluding user-defined constants.

2. The compile-time assignment statement may appear anywhere in the compilation unit.

12.1.3 COMPILE-TIME IF STATEMENT

The compile-time if statement is used to make the compilation of a piece of source code conditional upon the value of some boolean expression.

```
<compile-time if> ::= ? if <expression> ? then <text>
                      {? orif <expression> ? then <text>}
                      [? else <text>] ? ifend
```

The following rules apply:

1. The expression must be a boolean expression composed only of constants, and compile time variables, but excluding user-defined constants.

2. <text> is a string of source text which is compiled upon the value of the conditional expression. The text may contain compile time statements. Compile time statements are executed, and macro expansion takes place only in the

selected text.

3. The compile time if statement may appear anywhere in the compilation unit.

4. The text of a compile time if statement may contain either or both macro definitions or compile time variable definitions. In either case, the macro definition or the definition of the compile time variable will or will not be included in the compilation, depending upon the results of execution of the compile time if statement.

Example:

```
? var Table_size : integer := 50,
    Page_size : integer := 1024 ?;
var Table : array [1..Table_size] of integer
? if Table_size < 10 ? then

    "might include this procedure call into program."
    Bubblesort (Table)
? orif Table_size <= 2 * Page_size ? then

    "or call on procedure 'treesort' into program"
    Treesort (Table)
? else

    "choice 3, call on procedure Quicksort in program."
    Quicksort (Table)
? ifend
```

12.2 MACROS

A macro definition provides a string of source text which is compiled whenever the macro name is encountered by the compiler within the scope of the macro definition.

```
<macro definition> ::= macro <identifier>[(<macro parameter
    list>)]; <text> macroend
<macro parameter list> ::= <identifier>{,<identifier>}
```

The following rules apply:

1. The macro definition may appear anywhere in the compilation unit, and the scope of the macro name is from the point of declaration until a redeclaration of the same identifier as a macro.

2. <text> is a string of source text which effectively replaces any subsequent occurrences of the macro name within the scope of the macro. The text may contain compile-time statements and invocations of other macros, but these are evaluated at the time of substitution of the text -- not at the time of declaration. (A macro may invoke itself as a macro, but the invocation must be conditional so that when not executed, the nesting will cease in the compiler.)

3. All occurrences of formal parameters in the macro body are replaced by the corresponding actual parameters supplied by the macro invocation. The actual parameters are treated purely as symbol strings, and the parameter replacement has the effect of text substitution. Any macro names appearing in the actual parameters are invoked after the parameter has been substituted in the body.

   The actual parameter list follows the macro name and is surrounded by parentheses, and the parameters are separated by commas. Actual parameters may contain no unbalanced parentheses or brackets, and no commas except for those within parentheses or brackets.

4. The identifiers for a macro may not be the same as the identifiers of other parts of the compilation unit, including compile time variables. (See 6 below for exception.)

5. Macro definitions may not be statically nested (in source code) but one macro definition may contain an invocation of another disjointly defined macro (or of itself, see 3. above).

6. The identifiers of the formal parameters of a macro may be duplicated in other parts of the compilation unit (outside the macro definition) including use as another macro identifier. In the latter case it will be recognized as a formal parameter and not used to invoke the other macro, whenever it appears within the text of the macro definition in the source code.

Examples:

```
macro swap (a, b, type_id);
   begin
      var temp : type_id;
      temp := a;
      a := b;
      b := temp;
   end
macroend;
```

12.0 COMPILE-TIME FACILITIES
12.2 MACROS

```
                    " * * * * * "
   swap(x, y, array [1..10, 1..10] of integer) ;

                          " * * * * * "

macro forgn ; for g := 1 to n do macroend ;
macro forjn ; for j := 1 to n do macroend ;
macro forkn ; for k := 1 to n do macroend ;
var n : integer, elem : real ;
macro mac ; r[j,k] * s[k,j] macroend ;
     n := 27 ; "typical choice of array size"
     begin
     var r, s, t : array [1..n] of real ;
     var j, k, g : integer ;
lg:      forgn
           k := g ;
lj:        forjn
             elem := 0 ;
lk:          forkn
               elem := elem + mac
             forend ; "end of for loop lk"
             t[g,j] := elem
           forend "end of loop lj"
         forend "end of loop lg"
     end
```

13.0 REPRESENTATION-DEPENDENT FEATURES

13.0 REPRESENTATION-DEPENDENT FEATURES

In contrast to the previously discussed aspects of the language, the language features discussed in this section are data-representation dependent. This is not to imply that programs using these features are, in fact, system dependent. It is possible to write system independent programs through the careful use of representation-dependent features.

The representation-dependent features are such that use of these features may be dependent upon the compiler's allocation algorithms or the hardware design. The use of these features is, therefore, restricted to procedures declared with the repdep procedure attribute.

13.1 DATA TYPES

<rep type> ::= <cell type>
| <crammed type>

13.1.1 CELL TYPE

<cell type> ::= cell

A cell type is a representation-dependent basic type that represents the smallest unit of storage that is directly addressable by a pointer. A cell, therefore, has the aligned attribute.

Only the operations of assignment and equality test are defined on a cell.

13.1.2 CRAMMED TYPES

Through the use of crammed records and arrays, memory-dependent structures can be defined wherein the bit-size and alignment of each element is specified.

<crammed type> ::= [<alignment>] crammed <crammed structure>
<crammed structure> ::= <crammed array>|<crammed record>

13.0 REPRESENTATION-DEPENDENT FEATURES
13.1.2 CRAMMED TYPES

```
<crammed array> ::=
    array [<crammed indices>] of <crammed element>
<crammed indices> ::= <crammed index> {,<crammed index>}
<crammed index> ::= <fixed scalar type>
<crammed record> ::=
   record <crammed field>{,<crammed field>} recend
<crammed field> ::= <identifier list> : <crammed element>
<crammed element> ::=
      [<malignment>] <boolean type>
    | [<malignment>] <integer type>[<width>]
    | [<malignment>] <integer subrange type>[<width>]
    | <crammed type>
<malignment> ::= maligned (<offset>[,<base>])
<width>  ::= <integer constant>
<offset> ::= <integer constant>
<base>   ::= <integer constant>
```

13.1.2.1 Alignment

If a variable of crammed type is maligned, it will be allocated on an <offset> mod <base> (or an <offset>) cell boundary; otherwise, it is allocated on the first available cell boundary. If a crammed element is maligned, it will be allocated on an <offset> mod <base> bit boundary relative to the beginning of the innermost containing structure; if it is not maligned, it will be allocated at the next available bit. If <base> is not specified, the <base> of the containing structure is inherited.

Crammed variables can be passed by reference; crammed elements cannot.

Crammed types are equivalent if, and only if, they are equivalent in all conceivable ways.

Crammed types can conform to one another and to un-crammed types (c.f., Section 5.2.1).

13.1.2.2 Width

The width specification defines the bit size of the field.

Example:

```
type cyber_80_ptr =
            crammed record
"bit 0"             invalid : boolean
"bits 5 -14"        segment_num : maligned (5) integer [10]
"bits 35-63"        word_num : maligned (35) integer [28]
            recend
```

13.2 STATEMENTS

The result of an invocation of the loc function can be assigned to any direct pointer type (see 11.3.1).

14.0 MACHINE-DEPENDENT FEATURES

14.0 MACHINE-DEPENDENT FEATURES

An extended set of machine-dependent features including data types, storage attributes, machine instructions, and the code statement is provided each machine for which SWL will generate object code.

14.1 DATA TYPES

The only operations defined on machine-dependent data types are assignment and equality test.

Hypothetical example:

```
var  a : double,
     b : halfword,
     c : array [0..63] of byte,
     d : s#p#flt#pt,
     E : int#16,
     f : fl#pt,
     dfp : fl#pt#double ;
```

14.2 MACHINE-DEPENDENT STORAGE ATTRIBUTES

In general, the set of operations defined for a data type will not be affected by its storage attributes.

Hypothetical example:

```
var  tos : [register] halfword,
     param : [[illegible]] ^integer,
     accum : [[illegible] [3]] integer
```

14.3 CODE STATEMENT

The use of machine-dependent features is restricted to the body of the code statement.

<code statement> ::= code (<machine>)<code body> codend

14.0 MACHINE-DEPENDENT FEATURES
14.3 CODE STATEMENT

<code body> ::= <code>{,<code>}
<code> ::= <statement>
!<machine instruction>

14.4 MACHINE INSTRUCTIONS

The format of machine instructions may vary with the particular machine.

<machine instruction> ::= !<instruction>