Last modified on Tue Jun 26 11:24:34 PDT 1990 by ellis
modified on Thu Feb 16 10:34:09 PST 1989 by glassman
modified on Sat Jan 30 15:08:11 PST 1988 by gnelson

Modula-2+Epsilon Language Specification

Table of Contents

The Vulcan project of SRC is constructing a new Modula-2+ programming environment. We picked Modula-2+, not Modula-3, because our research goal is to provide important new functionality in a programming environment, not to explore a new language. We want Vulcan to be used by SRC programmers for their own programming, so we needed a way to accommodate the hundreds of thousands of lines of existing Modula-2+ code. Historically, programming-environment research projects that have attempted to provide both a new environment and a new langauge have ended up concentrating more on the new language and less on the environment. Since our time is limited and Modula-3 was (and still is) evolving, we decided that we could satisfy our goals for Vulcan by sticking with Modula-2+.

The original implementation of Modula-2+ was inadequate for Vulcan's goals, so we are reimplementing it from scratch. As a first step in the reimplementation, we wanted a concise langauge definition. Looking at the original Modula-2+ reference manual, we realized that though most SRC programmers viewed the language as fairly small and simple, in fact there were many warts and barnacles that caused the reference manual (written after the fact) to be large and complicated. In practice, the only reliable definition of the language was the VAX Modula-2+ compiler, which sometimes differed from the debugger and other tools over the meaning of the language.

Since we are reimplementing the language, we decided to take the opportunity to scrape off many of the barnacles, resulting in a simpler language and a (hopefully) simpler implementation. We called the resulting language Modula-2+Epsilon. To the practicing programmer at SRC, there is very little difference between Modula-2+ and Modula-2+Epsilon. We expect to port over a million lines of Modula-2+ into Modula-2+Epsilon with very little effort.

But for the Vulcan implementors, Modula-2+Epsilon is quite a bit simpler. The most important change was simplifying the type system by adapting a subset of Modula-3's type system. Instead of Modula-2+'s long list of fairly arbitrary rules defining the separate type relations of "compatibility", "assignability", and "passability", there is now a reasonably short definition of a single relation, "assignability".

We scraped off numerous other barnacles as well. The ground rules for changing the language were:

> We wanted a high degree of compatibility with the existing programs written in Modula-2+.
>
> An old feature could be deleted or modified if it didn't affect more than a few existing programs.
>
> An old feature could be modified or a new feature could be added only if it would clearly simplify the Vulcan implementation or if it would produce clearly identifiable large benefits to SRC programmers (inline procedures is the only new feature in the latter category).

There are many features remaining in the langauge that many of us don't particularly like. But since those features are not directly relevant to the goals of Vulcan, we resisted the urge to fiddle.

To produce this definition, I started with an early draft of the Modula-3 langauge definition, written by Greg Nelson and Lucille Glassman. They bear no responsibility for the contents of this document.

-- John R. Ellis

1. Overview

This text is the definition of Modula-2+, a general-purpose programming language intended mainly for constructing large systems. Modula-2+ was derived from Modula-2 [1, 2]. The main differences from Modula-2 are:

References (pointers), which the programmer can declare to be garbage-collected or uncollected.

Exceptions, which can exit many levels of procedure calls, namely those levels between the point at which the exception is raised and the point at which it is handled.

Concurrency and synchronization primitives, in the form of mutual exclusion semaphores and condition variables.

This document includes some examples and expository prose, but it is a language definition, not a tutorial or reference manual. It is an operational English definition rather than an axiomatic semantic definition, and is intended for a wide audience.

The rest of the overview introduces some fundamental concepts of the language.

1.1. Variables

A *variable* is a component of the computational state. Some variables are named by identifiers; others, like the components of arrays, are denoted by computed expressions. A variable in another module is named by a *qualified identifier*, which is an identifier preceded by a module name, as in `Text.Equal` or `Thread.Fork`.

A *constant* is a name for a value that can be computed at compile time.

1.2. Types

Every variable has a unique type, which is specified when the variable is declared. For every type, the language specifies the *value set* (the set of values that variables of the type can represent) but leaves the choice of representation to the implementation.

Although every variable has a unique type, a *value* may not have a unique type, since the value sets of different types can overlap. For example, the value `6` has type `INTEGER` and type `[0..9]`, but the expression `6` has static type `INTEGER`. Thus the phrase "type of x" may be ambiguous. By the "syntactic type of x" we mean the declared type of the variable x or the statically determined type of the expression x.

As in Pascal, distinct occurrences of identical type expressions denote different types.

The type denoted by a type name is fixed and usually known at compile time. But if the declaration and the denotation appear in different scopes, then the type denoted by a name is not known at compile time. A type name whose denotation is unknown at compile time is said to be *opaque*; otherwise the type name is *concrete*. The use of opaque types minimizes compilation dependencies, since a compilation that is independent of the denotation of a type name need not be repeated when the denotation changes.

1.3. Values

Modula-2+'s value space is similar to that of languages such as Pascal and C, except that Modula-2+ also provides garbage-collected references. Here are the possible values that variables can assume:

An *integer* value represents an element of an implementation-dependent subrange of the mathematical integers. Integers are implemented using the natural word-size of the machine.

An *unsigned* value represents an element of an implementation-dependent subrange of the non-zero mathematical integers. Unsigneds are implemented using the natural word-size of the machine.

An *enumeration* is an ordered collection that the programmer defines by listing its elements.

Integers, unsigneds, and enumerations are collectively referred to as *ordinal values*. Any type whose values are ordinal values is an *ordinal type*.

A *floating point* number is an approximate representation of a real number; the details of the representation are implementation-dependent. Modula-2+ provides both single-precision and double-precision floating point values, which are distinct from one another.

A *set* is a collection of values taken from some ordinal type.

A *reference* `r` is either `NIL` or the address of a garbage-collected variable `r^`. The implementation must store the referent in a system-managed storage pool, be able to determine at runtime when all references to it are gone, and be able to reclaim the storage for it. It must also be able to determine the type of the referent at runtime.

A *pointer* `r` is either `NIL` or the address of a variable `r^` that may or may not be collected. Pointer values are created by allocating new variables from a system-managed storage pool, by taking the address of another variable, or by performing address arithmetic. The programmer is responsible for reclaiming the storage of pointer referents.

An *array* `a` is an indexed collection of variables of the same type; `a[i]` denotes the variable associated with index `i`. The indexes must be of an ordinal type, called the *index type* of the array. The variables all have the same size and the same type, called the *element type* of the array.

A *record* `r` is a collection of named variables; `r.f` denotes the variable associated with the field `f` in record `r`. Different fields can have different types. There are *fixed* records and *variant* records: in fixed records the set of fields cannot change; in variant records there are one or more *tag fields* whose value determines the set of fields the record contains.

A *procedure* is an executable code sequence. A procedure must have a name, and can optionally take parameters and return a result. There are four parameter modes for passing arguments: value (the default), `VAR`, `VAR IN`, and `VAR OUT`. A procedure value also has an associated set of exceptions, called the *raises set* for the procedure, which specifies the exceptions that the procedure can raise.

A procedure that returns a result is called a *function procedure*; a procedure that does not return a result is called a *proper procedure*. A *local procedure* is

a procedure declared within the scope of another procedure.

1.4. Errors and safety

A *static error* is an error that implementations are required to detect before program execution; implementations are required to reject any program containing static errors.

A *checked runtime error* is an error that implementations are required to detect and report at runtime or earlier. The method for reporting them is implementation-dependent.

An *unchecked runtime error* is an error that is not guaranteed to be detected, and can thus cause the subsequent behavior of the computation to be arbitrary. Implementations may detect and report some unchecked runtime errors.

Unchecked runtime errors arise only from the use of certain features intended primarily for low-level programming, such as explicit storage deallocation and loopholing. Any module not using these features can be labeled *safe*. In a safe module, the implementation guarantees the absence of unchecked runtime errors; in an unsafe module, it is the programmer's responsibility to avoid them. See 6.5, page 45.

1.5. Required interfaces

An implementation of Modula-2+ must include implementations of the interfaces Text, Rd, RdV, Wr, WrV, and Thread. These interfaces are not described here; SRC's versions are described in "The Rubberneck Tour of the Public Interfaces".

Text provides elementary operations on text strings. `Text.T` is the type of a text string literal; it is a ref type.

Rd, RdV, Wr, and WrV provide operations on streams of characters. `Rd.T` is the type of a reader stream and `Wr.T` is the type of a writer stream; they are ref types. `PRINTF` requires a `Wr.T` or `WrV.T`, and `SCANF` requires an `Rd.T` or `RdV.T` (5.23, page 38).

Thread provides operations on concurrent threads of control (*lightweight processes*). `Thread.Mutex` is the type of a mutual-exclusion lock. The `LOCK` statement requires a `Thread.Mutex`.

2. Types

2.1. Atomic types

Here are the atomic types, which correspond to the different kinds of scalars in the value space:

INTEGER	an integer value
UNSIGNED	an unsigned value
REAL	a single-precision floating point value
LONGREAL	a double-precision floating point value
Null	the value **NIL**
BYTE	a machine byte (same size as **CHAR**)
WORD	a machine word (same size as **INTEGER**)

Unfortunately, the language does not have a predefined identifier naming the type Null.

Modula-2+ also provides the following atomic types, which could be defined in the language but are built-in for convenience:

BITSET	`SET OF [0..31]`
BOOLEAN	The enumeration **(FALSE, TRUE)**
CARDINAL	The subrange `[0..LAST(INTEGER)]`
CHAR	A 256-element enumeration, of which the first 128 elements are the ASCII character set

2.2. Constructed types

This section introduces each of Modula-2+'s type constructors and gives an informal syntactic description for each. For the complete syntax, see Appendix I, page 49.

2.2.1. Enumerations

If `IdList` is a list of identifiers, then:

```
TYPE T = ( IdList )
```

constructs a new enumeration type **T** with one value for each identifier in the list, in the same order as the list. The identifiers in `IdList` become constants that name the values of **T**, whose scope is the same as the scope of **T**. For example:

```
TYPE Color = (Red, Green, Blue)
```

defines the names **Red**, **Green**, and **Blue** as well as `Color`. Empty enumerations are allowed.

2.2.2. Subranges

If **Lo** and **Hi** are integers, unsigneds, or two values from the same enumeration, then:

```
TYPE T = [Lo..Hi]
```

constructs a new subrange type **T**. The values of **T** are all the values that lie between **Lo** and **Hi** inclusive. **Lo** and **Hi** must be constant expressions (3.3, page 24). It is legal for **Lo** to exceed **Hi**, in which case the subrange is empty.

If **Lo** and **Hi** are numbers, then if either is unsigned, **T** is a subrange of **UNSIGNED**, and **Lo** must be non-negative. Otherwise, **T** is a subrange of **INTEGER**.

Note that each enumeration type introduces a new collection of values, but a subrange of an enumeration type reuses the values from the underlying type.

2.2.3. Arrays

There are two kinds of array types, *fixed* array types and *open* array types. If **Index** is an ordinal type and **Element** is any type other than an open array type, then:

```
TYPE T = ARRAY Index OF Element
```

constructs a new fixed array type **T**. The values of type **T** are arrays whose element type is **Element** and whose length is the number of elements of the type **Index**. If **a** has type **T**, then **a[i]** denotes the element of **a** whose position corresponds to the position of **i** in **Index**.

Multi-dimensional arrays are allowed; a declaration of the form:

```
ARRAY Index1, Index2, ..., Indexn OF Element
```

is an abbreviation for the construction:

```
ARRAY Index1 OF ARRAY Index2 OF ... ARRAY Indexn OF Element
```

If **Element** is any type, then:

```
TYPE T = ARRAY OF Element
```

constructs an open array type **T**. The values of **T** are arrays whose element type is **Element** and whose length is arbitrary. The index set of an open array with **n** elements is **[0..n-1]**.

An open array type can be used either as the referent of a **REF** type or as the type of a formal parameter. An open array type cannot be used as the type of a variable declaration, a record field, an array element, or a procedure result.

When an open array type is used as the referent of a **REF**, the size of the array is determined by the arguments passed to **NEW** when the array variable is allocated (see 5.19, page 37).

When an open array type is used as a procedure formal, the actual array can be either a fixed array or an open array; the element type of the actual must be equivalent to the element type of the formal. The procedure can use **HIGH** or **NUMBER** (page 23) to determine the size of the actual. The interpretation of array indexes depends on the syntactic type of the array: when an actual array **a** is bound to an open array formal **b**, the index set of **a** is translated to start at 0. Thus, for example, **b[0]** denotes the first element of **a**.

2.2.4. Records

There are two kinds of record types, *fixed* record types and *variant* record types. If $\texttt{field}_1, \ldots, \texttt{field}_n$ are distinct identifiers and $\texttt{T}_1, \ldots, \texttt{T}_n$ are any types, a declaration of the form:

```
TYPE T = RECORD
  field1: T1;
    ...
  fieldn: Tn;
END
```

constructs a new fixed record type `T`. The values of type `T` are records with fields $\texttt{field}_1, \ldots, \texttt{field}_n$ of types $\texttt{T}_1 \ldots \texttt{T}_n$. If `r` has type `T`, then `r.f` denotes the field named `f` in the record `r`.

When a series of fields shares the same type, $\texttt{field}_i$ can be a list of identifiers separated by commas:

```
f1, ..., fn : type
```

This is shorthand for:

```
f1 : type;
  ...
fn : type;
```

In a variant record declaration, one or more of the field lists are replaced by variant sections. A variant section has the form:

```
CASE tag: TagType OF
| LabelList1: FieldList1
| ...
| LabelListn: FieldListn;
  ELSE FieldList0;
END;
```

where `TagType` is an ordinal type, each `LabelList` is a list of ranges of constants of that type, and each `FieldList` is a field list. No value can appear in more than one `LabelList`. The names in a field list must be distinct from one another, from the names in all other field lists defined in the record, and from `tag`.

A variant record `r` whose type includes the variant section above contains a field `r.tag` of type `TagType`. If the value of `r.tag` appears in $\texttt{LabelList}_i$, then `r` also contains the fields listed in $\texttt{FieldList}_i$. If the value of `r.tag` does not appear in any of the label lists, then `r` contains the fields listed in the `ELSE` clause. If there is no `ELSE` clause, `r` contains no additional fields.

The contents of a variant record are always interpreted according to the current value of the tag field. It is a checked runtime error to access a field outside the field list for the current tag value. Safe modules can change the tag of a variant record or assign variant records only if the variant fields of the record type are representation complete (2.5, page 13, 6.5, page 45). The effect of reading a field after changing the tag is implementation-dependent.

The tag "`tag :`" may be omitted from a variant section, yielding a tagless variant. Any field of the variant section may be read or written at any time. A tagless variant record type can be used in a safe module only if all of its variant fields are representation complete (2.5, page 13, 6.5, page 45).

Variant sections are represented by a block of storage large enough for any of the

variants, and the representations for the fields are overlaid in the block.

Empty records are allowed.

2.2.5. References

If **Type** is any type, then:

```
TYPE T = REF Type
```

constructs a new reference type **T**. The values of **T** are garbage-collected references to variables of type **Type**. The type **T** is called a *ref type*.

A unique *type code* is assigned to every ref type, and all referents of a ref type are tagged with the same type code; see **TYPECODE**, 3.2.7, page 23.

Omitting **Type** constructs an *opaque ref type*:

```
TYPE T = REF
```

See 4.2, page 26.

In the safe part of the language, the only way to generate a reference value is by calling **NEW** (5.19, page 37).

The type:

```
TYPE T = REFANY
```

includes all references to variables of all ref types.

A type is *ref containing* if it is a ref type, **REFANY**, a record type with a ref-containing field, or an array with a ref-containing element type.

2.2.6. Pointers

If **Type** is any type, then:

```
TYPE T = POINTER TO Type
```

constructs a new pointer type **T**. The values of **T** are addresses of variables of type **Type**. The type **T** is called a *pointer type*.

See 4.2, page 26 for constructing opaque pointer types.

New pointer values can be created by calling **NEW** (5.19, page 37) to create a new variable, or in unsafe modules, by taking the address of a variable with **ADDR** or by address arithmetic.

The type:

```
TYPE T = ADDRESS
```

includes all references to variables of all pointer types.

2.2.7. Sets

If **Type** is any ordinal type, then:

```
TYPE T = SET OF Type
```

constructs a new set type **T**. The values of **T** are all sets whose elements have type **Type**. For example, a variable whose type is **SET OF [0,1]** can assume the following values:

```
{}    {0}    {0,1}    {1}
```

2.2.8. Procedures

A procedure type declaration has the form:

```
TYPE T = PROCEDURE(
  mode1 type1 := default1,
  ...
  moden typen := defaultn ) : R
RAISES S
```

where:

Each **mode**$_i$ is **VAR**, **VAR IN**, **VAR OUT** or empty, which means by value.

Each **name**$_i$ is an identifier, which names the formal parameter.

Each **type**$_i$ is a qualified identifier denoting a type, or else an open array type expression.

Each **default**$_i$ is a constant expression. The "**:= default**" is optional for **VAR IN** and by-value parameters, and not allowed for **VAR** or **VAR OUT** parameters. If parameter **i** has a default, then so must parameters **i+1** through **n**.

R is a qualified identifier denoting any type other than an open array type. "**: R**" is optional.

S is a list of qualified identifiers denoting exceptions, enclosed in curly braces. **RAISES S** is optional; if missing, it defaults to the set of all exceptions.

The specification of parameters, result, and **RAISES** clause is called the *procedure signature*.

A procedure **P** has type **T** defined above if:

P takes **n** parameters, whose types are **type**$_1$, ..., **type**$_n$.

P is a function procedure whose result type is **R** or **P** is a proper procedure and the "**: R**" was omitted.

Every exception in **P**'s raises set appears in the set **S**.

Parameter defaults do not affect the set of procedures of type **T**.

It is a checked runtime error for a procedure to raise an exception not listed in its **RAISES** clause.

2.2.9. Bits for

If **Type** is any non-ref-containing type and **n** is a constant expression, then:

```
TYPE T = BITS n FOR Type
```

constructs a new type **T**. The values of type **T** are the same as the values of type **Type**. But variables of type **T** will occupy exactly **n** bits if they appear within records or arrays. For example:

```
TYPE T = ARRAY [0..255] OF BITS 1 FOR BOOLEAN;
```

constructs an array of 256 booleans that requires 256 bits of storage.

The values allowed for **n** are implementation-dependent, but **n** must not be less than the minimum number of bits needed to represent the value set. Illegal values for **n** generate a static error.

2.3. Same type

Two identifiers **U** and **V** denote the same type if **U** and **V** are the same identifier, or if **U** is defined as:

```
TYPE U = W
```

where **W** denotes the same type as **V**. This implies that every occurrence of the subrange constructors, enumeration constructors, and the constructors **REF**, **POINTER**, **RECORD**, **ARRAY**, **SET**, and **BITS** denotes a different type. For example:

```
TYPE
    U = REF INTEGER;
    V = REF INTEGER;
```

U and **V** denote different types.

2.4. Assignability

If type **U** is *assignable* to type **V**, then a value of type **U** can be assigned to a variable of type **V** or passed to a formal by-value parameter of type **V**. Intuitively, two types are assignable if their value sets intersect. More formally, **U** is assignable to **V** if:

U and V are the same type.

U is a subrange of T and T is assignable to V, or vice versa.

U is **UNSIGNED** and V is **INTEGER**, or vice versa.

U is **REFANY** and V is **REF T**, or vice versa.

U is **ADDRESS** and V is **POINTER TO T** (in unsafe modules only), or U is **POINTER TO T** and V is **ADDRESS**.

U is Null and V is a **REF**, **REFANY**, **POINTER**, **ADDRESS**, or **PROCEDURE** type.

U is defined as **BITS n FOR T** and T is assignable to V, or vice versa.

U is **ARRAY I OF S**, V is **ARRAY J OF T**, and S and T are the same type; U or V or both may be open arrays.

U and V are procedure types with the same number of parameters, corresponding parameters have the same type or are open arrays of the same type, corresponding parameters are declared with the same mode, the results have the same type, and the raises set of U is a subset of the raises set of V (a missing raises set defaults to the set of all exceptions).

BITSIZE(U) = BITSIZE(BYTE), U is not an open array, and V is **BYTE**.

BITSIZE(U) = BITSIZE(WORD), U is not an open array, and V is **WORD**.

BITSIZE(U) MOD BITSIZE(BYTE) = 0 and V is **ARRAY OF BYTE**.

BITSIZE(U) MOD BITSIZE(WORD) = 0 and V is **ARRAY OF WORD**.

2.5. Unsafe types

A type T is *representation complete* if any pattern of b bits represents a valid value of T, where **b = BITSIZE(T)**. (This is obviously implementation dependent.) In safe modules, some operations such as **LOOPHOLE** are restricted to representation-complete types.

In the current implementation, T is representation complete if it is not an opaque type and:

T is **INTEGER**, **UNSIGNED**, **REAL**, **LONGREAL**,
CHAR, **BYTE**, **WORD**, **ADDRESS**;

T is an enumeration, and **NUMBER**(**T**) = 2^b;

T is a subrange, and either **FIRST**(**T**) = 0 and **LAST**(**T**) = 2^b-1,
or **FIRST**(**T**) = -2^(b-1) and **LAST**(**T**) = 2^(b-1)-1;

T is a record and the types of the fields of **T** are
representation complete;

T is an array and the element type of **T** is representation
complete;

T is a set.

Types that contain refs, pointers, or procedures are not representation complete.

An *unsafe type* is one whose use may produce unchecked runtime errors. A type **T** is unsafe if:

T is a pointer to a ref-containing or unsafe type;

T is a ref to an unsafe type;

T is a record type with at least one unsafe field type;

T is a tagless variant record with at least one variant field
that is not representation complete;

T is an array type whose element type is unsafe;

T is a **BITS FOR** of an unsafe type.

T is a **PROCEDURE** type whose result type is unsafe.

The use of unsafe types is not allowed in safe modules (6.5, page 45).

3. Expressions

An expression consists of operands and operators applied to other expressions. The operands can be qualified identifiers, literals, or expressions. Every expression has a syntactic type, which the implementation can determine from the rules in this chapter.

Expressions that can denote variables as well as values are called *designators*. There are certain contexts, such as assignment statements, **VAR** parameters, **WITH**, **NEW**, **INC**, and **DEC**, which require designators rather than arbitrary expressions. When a designator is used in such a context, it denotes the variable; when used in other contexts, it denotes the value of the variable.

Any qualified identifier declared as a variable is a designator. If `d` is a designator and `e` is an expression, then the following are also designators:

```
e^
d[e]
d.e
(d)
```

3.1. Literals

A *literal* is an expression, like a numeral or text string, that directly represents a constant value. The type of a literal can always be determined from its denotation.

3.1.1. Numeric literals

Literal numbers represent nonnegative integers, unsigneds, or reals. A literal integer or unsigned is a sequence of digits with an optional suffix to indicate the base. A literal (decimal) real is a sequence of digits and optional scale factors. Case is not significant in literal digits, prefixes or scale factors.

The type of a non-real literal number is **INTEGER** if it is no larger than **LAST(INTEGER)**; otherwise the type is **UNSIGNED**.

A literal decimal **INTEGER** or **UNSIGNED** uses the digits `0` through `9`.

A literal octal **INTEGER** or **UNSIGNED** uses the digits `0` through `7` and is suffixed by `B`.

A literal hex **INTEGER** or **UNSIGNED** uses the digits `0` through `9` and **A** through **F** and is suffixed by `H`. To avoid ambiguity, hex numbers must begin with `0` through `9`.

A literal **REAL** uses the decimal digits and can include a scale factor preceded by `E` or `e`. A literal **LONGREAL** is indicated by the scale factor `D` or `d`; a missing scale factor implies **REAL**. Examples:

```
12.3    45.67E-8    0.8d    0.8d-10
```

The type of the first two examples is **REAL**; the type of the second two is **LONGREAL**.

3.1.2. Text and character literals

A literal **CHAR** is a single ASCII character or an escape sequence enclosed in single quotes.

A text literal is a sequence of zero or more characters or escape sequences enclosed in double quotes. The syntactic type of a text literal is **Text.T**.

The escape sequences for characters and texts are:

```
\n  newline (linefeed)          \b  backspace
\t  tab                         \\  backslash
\r  carriage return             \"  double quote
\f  form feed                   \'  single quote
\<newline> nothing
```

A \ followed by exactly three octal digits specifies the character whose ASCII code is that octal value. A \ followed by anything other than an escape sequence or an octal value is a static error. A \ followed by a newline specifies nothing -- \<newline> lets a text span multiple lines.

3.1.3. NIL

The literal "**NIL**" denotes the value **NIL**. Its syntactic type is Null.

3.1.4. Set literals

If **S** is a set type and e_1, ..., e_n are constant expressions assignable to the element type of **S**, then an expression of the form:

S{e_1**,** ... **,** e_n**}**

represents a constant value of set type **S** containing the listed elements. Order is not important. An e_i can have the form **lo** **..hi**, which denotes the range of elements between **lo** and **hi**, inclusive. **S{}** denotes the empty set.

3.2. Operators and operations

The rest of this section describes each operator and operation in detail. Since the signatures of some operators cannot be described in ordinary Modula-2+, we adopt the following conventions:

For *overloaded* operations (the same symbol represents several different operations and the actual operation is determined by the types of the arguments), there is a separate definition for each possible operand type. For example:

```
ABS(x: INTEGER)  : INTEGER;
   (x: REAL)     : REAL;
   (x: LONGREAL) : LONGREAL
```

overloads the **ABS** operator with three different types. The syntactic type of the result is determined by the syntactic type of the operands. For operations involving both **INTEGER** and **UNSIGNED** operands (or subranges of those types), the **UNSIGNED** overloading is used. (For example, adding an integer to an unsigned yields an unsigned.)

For operations that take any array, or any set, or the like, only the first letter of the type name is capitalized. For example:

```
IN(e: Element; s: Set): BOOLEAN;
```

For operations on types, the word **Type** appears in the heading:

```
FIRST(T: Type): T;
```

Unless explicitly stated otherwise, operators behave just like normal function procedures with regard to parameter passing and return values (5.15, page 35).

3.2.1. Precedence and order of evaluation

Here are the classes of operator precedence, from highest to lowest:

```
f(x) a[i] p^ x.y      (application, subscript, dereference,
                       record/module selection)
+  - NOT              (unary ops)
* / DIV MOD AND       (multiplicative ops)
+ - OR                (additive ops)
= # < <= >= > IN      (predicates)
```

All operators are left associative. The operands in **AND** and **OR** operations are evaluated from left to right. The order of evaluation for the operands of other operations is undefined; procedure actuals can be evaluated in any order. Parentheses can be used to introduce grouping.

3.2.2. Function procedures

A procedure name followed by a (possibly empty) parenthesized parameter list denotes a call on the procedure. The syntactic type of the expression is the return type of the procedure. A procedure name that is not followed by a parenthesized list denotes the procedure itself.

The value of the expression is the value returned by the procedure. See 5.15, page 35 and 5.16, page 36 for the semantics of procedure call and return.

3.2.3. Arithmetic operations

The effect of a numeric operation that overflows, underflows, or divides by zero is implementation-dependent. The effect can be either a checked runtime error or an erroneous answer. However, operations on unsigneds never cause a runtime error.

Address arithmetic (operations involving addresses and integers) is performed in the natural addressing units of the underlying machine.

```
+ (x: INTEGER)  : INTEGER;
  (x: UNSIGNED) : UNSIGNED;
  (x: REAL)     : REAL;
  (x: LONGREAL) : LONGREAL;
  (x: ADDRESS)  : ADDRESS;
  (x: Set)      : Set;

+ (x,y: INTEGER)                 : INTEGER;
  (x,y: UNSIGNED)                : UNSIGNED;
  (x,y: REAL)                    : REAL;
  (x,y: LONGREAL)                : LONGREAL;
  (x: ADDRESS; y: INTEGER)       : ADDRESS;
  (s,t: Set)                     : Set;
```

When used as a unary prefix operator, means identity.

When used as a binary infix operator on numeric operands, returns the sum of `x` and `y`.

For sets `s` and `t`, returns set union. That is:

```
x IN (s + t)  iff  (x IN s) OR (x IN t)
```

The set `s` must be assignable to `t` or vice versa.

```
- (x: INTEGER)                   : INTEGER;
  (x: UNSIGNED)                  : UNSIGNED;
  (x: REAL)                      : REAL;
  (x: LONGREAL)                  : LONGREAL;
  (x: Set)                       : Set;

- (x,y: INTEGER)                 : INTEGER;
  (x,y: UNSIGNED)                : UNSIGNED;
  (x,y: REAL)                    : REAL;
  (x,y: LONGREAL)                : LONGREAL;
  (x: ADDRESS; y: INTEGER)       : ADDRESS;
  (x,y: ADDRESS)                 : INTEGER;
  (s,t: Set)                     : Set;
```

When used as a unary operator on numeric operands, means negation ("`0-x`").

When used as a binary operator on numeric operands, returns the difference of `x` and `y`.

When used as a unary operator on a set, returns the complement of the set (all elements not in the set).

When used as a binary operator on sets, returns the set difference. That is:

```
x IN (s - t)  iff  (x IN s) AND NOT (x IN t)
```

The set `s` must be assignable to `t` or vice versa.

```
* (x,y: INTEGER) : INTEGER;
  (x,y: UNSIGNED): UNSIGNED;
  (x,y: REAL)     : REAL;
  (x,y: LONGREAL): LONGREAL;
  (s,t: Set)      : Set;
```

When used as a binary infix operand on numeric operands, returns the product of **x** and **y**.

When used on sets, returns the intersection. That is:

```
x IN (s * t)    iff  (x IN s) AND (x IN t)
```

The set **s** must be assignable to **t** or vice versa.

```
/ (x,y: REAL)     : REAL;
  (x,y: LONGREAL): LONGREAL;
  (s,t: Set)      : Set;
```

When used on numeric operands, performs real division. On set operands, performs symmetric set difference. That is:

```
x IN (s / t)    iff  (x IN s) # (x IN t)
```

The set **s** must be assignable to **t** or vice versa.

```
DIV (x,y: INTEGER) : INTEGER;
    (x,y: UNSIGNED): UNSIGNED;

MOD (x,y: INTEGER) : INTEGER;
    (x,y: UNSIGNED): UNSIGNED;
```

x DIV y is the quotient of **x** and **y** truncated towards zero.

x MOD y is defined to be **x - y * (x DIV y)**.

```
ABS (x: INTEGER) : INTEGER;
    (x: REAL)     : REAL;
    (x: LONGREAL): LONGREAL
```

ABS returns the absolute value of x; overflow is possible.

```
FLOAT (x: INTEGER) : REAL;
      (x: UNSIGNED): REAL;
      (x: REAL)     : REAL;
      (x: LONGREAL): REAL;

LONGFLOAT (x: INTEGER) : LONGREAL;
          (x: UNSIGNED): LONGREAL;
          (x: REAL)     : LONGREAL;
          (x: LONGREAL): LONGREAL;
```

FLOAT converts **x** into the nearest **REAL**; **LONGFLOAT** converts **x** into the nearest **LONGREAL**.

```
ROUND (r: REAL)     : INTEGER;
      (r: LONGREAL): INTEGER;

TRUNC (r: REAL)     : INTEGER;
      (r: LONGREAL): INTEGER;
```

ROUND rounds **r** to the nearest integer value; **TRUNC** converts **r** to the integer value that is closer to zero than **r** is. Thus, for real operands **ROUND** is equivalent to:

```
IF (x >= 0.0) THEN
  RETURN TRUNC(x + 0.5);
ELSE
  RETURN -TRUNC(-x + 0.5);
END;
```

The details of "nearest" are implementation-dependent.

```
FLOOR   (x: REAL)     : INTEGER;
        (x: LONGREAL) : INTEGER;

CEILING (x: REAL)     : INTEGER;
        (x: LONGREAL) : INTEGER;
```

FLOOR(x) is the largest integer not greater than **x**. **CEILING(x)** is the smallest integer not less than **x**.

```
MAX, MIN (x,y: INTEGER)    : INTEGER;
         (x,y: UNSIGNED)   : UNSIGNED;
         (x,y: REAL)       : REAL;
         (x,y: LONGREAL)   : LONGREAL;
         (x,y: Enumerated) : Enumerated;
```

MAX returns the greater of the two values **x** and **y**; **MIN** returns the lesser.

3.2.4. Relations

```
=, # (x,y: T): BOOLEAN;
```

where **T** is one of **INTEGER**, **UNSIGNED**, **REAL**, **LONGREAL**, **Enumerated**, **Ref**, **ADDRESS**, **Procedure**, Null, **Set**, **Array**, or **Record**.

= is a binary infix operator that returns **TRUE** if **x** and **y** have the same value. # is a binary infix operator that returns **TRUE** if **x** and **y** have different values.

References have the same value if they address the same variable. Sets and arrays are the same if they have the same elements. Records are the same if the values of their fields are the same.

The syntactic type of **x** must be assignable to the syntactic type of **y**, or vice versa.

```
>, < (x,y: T): BOOLEAN;
```

where **T** is one of **INTEGER**, **UNSIGNED**, **REAL**, **LONGREAL**, **Enumerated**, **ADDRESS**, or **Set**.

< is a binary infix operator that returns **TRUE** if **x** is less than **y**. On enumerations, returns **TRUE** if the ordinal value of **x** is less than the ordinal value of **y**. On sets, returns **TRUE** if **s** is a proper subset of **t**; that is:

```
s < t    iff    (s <= t) AND (s # t)
```

> is a binary infix operator that returns **TRUE** if **x** is greater than **y**.

The syntactic type of **x** must be assignable to the syntactic type of **y**, or vice versa.

```
<=, >= (x,y: T): BOOLEAN;
```

where **T** is one of **INTEGER**, **UNSIGNED**, **REAL**, **LONGREAL**, **Enumerated**, **ADDRESS**, or **Set**.

<= is a binary infix operator that returns **TRUE** if **x** is not greater than **y**. On set operands, returns **TRUE** if every element of **s** is an element of **t**. On

enumerations, returns **TRUE** if the ordinal value of **x** is not greater than the ordinal value of **y**.

>= is a binary infix operator that returns **TRUE** if **x** is not less than **y**.

The syntactic type of **x** must be assignable to the syntactic type of **y**, or vice versa.

```
IN (e: Element; s: Set): BOOLEAN;
```

IN is a binary infix operator that returns **TRUE** if **e** is an element of the set **s**. **Element** must be assignable to the element type of **s**.

3.2.5. Boolean operations

```
NOT (p: BOOLEAN): BOOLEAN;
```

NOT p is the logical complement of the boolean expression **p**.

```
AND (p,q: BOOLEAN): BOOLEAN;
```

p AND q has the value **TRUE** if and only if both **p** and **q** are **TRUE**. If **p** is **FALSE**, **q** is not evaluated.

```
OR (p,q: BOOLEAN): BOOLEAN;
```

p OR q is **TRUE** if at least one of **p** or **q** is **TRUE**. If **p** is **TRUE**, **q** is not evaluated.

3.2.6. Bit operations

The bit operations treat the representations of integer and unsigned values as vectors of w bits, where w is the natural word size of the machine. The bits are numbered 0 to w-1 from right to left, where bit 0 is the least significant bit.

```
BITNOT (x: INTEGER  ): INTEGER
       (x: UNSIGNED ): UNSIGNED
```

Returns the bitwise complement of **x**.

```
BITAND (x, y: INTEGER  ): INTEGER
       (x, y: UNSIGNED ): UNSIGNED

BITOR  (x, y: INTEGER  ): INTEGER
       (x, y: UNSIGNED ): UNSIGNED

BITXOR (x, y: INTEGER  ): INTEGER
       (x, y: UNSIGNED ): UNSIGNED
```

Returns the bitwise **AND**, **OR**, or exclusive **OR** of the arguments.

```
BITSHIFTLEFT  (x: INTEGER;  n: INTEGER ): INTEGER
              (x: UNSIGNED; n: INTEGER ): UNSIGNED

BITSHIFTRIGHT (x: INTEGER;  n: INTEGER ): INTEGER
              (x: UNSIGNED; n: INTEGER ): UNSIGNED
```

BITSHIFTLEFT returns **x** shifted left by **n MOD BITSIZE(INTEGER)** bits, with zeros inserted at the right (least significant) end. **BITSHIFTRIGHT** returns **x** shifted right by **n MOD BITSIZE(INTEGER)**] bits, with zeros inserted at the left (most significant) end.

```
BITROTATELEFT  (x: INTEGER;  n: INTEGER): INTEGER
               (x: UNSIGNED; n: INTEGER): UNSIGNED

BITROTATERIGHT (x: INTEGER;  n: INTEGER): INTEGER
               (x: UNSIGNED; n: INTEGER): UNSIGNED
```

BITROTATELEFT returns the bits of **x** rotated left by **n MOD BITSIZE(INTEGER)** bits. **BITROTATERIGHT** returns the bits of **x** rotated right by **n MOD BITSIZE(INTEGER)** bits.

```
BITEXTRACT (x: INTEGER;  o, n: INTEGER ): INTEGER
           (x: UNSIGNED; o, n: INTEGER ): UNSIGNED
```

The least significant bits of the result are bits **o** through **o+n-1** of **x**; the other bits of the result are 0. It is a checked runtime error if **o+n > BITSIZE(INTEGER)**, **n < 0**, **o < 0**, or **o > BITSIZE(INTEGER)**.

```
BITINSERT (x: INTEGER;  o, n: INTEGER; y: INTEGER  ): INTEGER
          (x: UNSIGNED; o, n: INTEGER; y: UNSIGNED ): UNSIGNED
```

The result is **y** with bits **o** through **o+n-1** replaced by the least significant **n** bits of **x**. It is a checked runtime error if **o+n > BITSIZE(INTEGER)**, **n < 0**, **o < 0**, or **o > BITSIZE(INTEGER)**.

3.2.7. Type operations

```
ORD (element: Ordinal): INTEGER;
VAL (T: OrdinalType; e: INTEGER): T;
```

ORD returns the position of an element within its type's enumeration order. If the syntactic type of **element** is a subrange of type **T**, the result is the position of the element within **T**, not the subrange. The first value in any enumeration type has ordinal value 0.

VAL is the inverse of **ORD**; it converts from a position **e** into the element that occupies that position in the enumeration. If **T** is a subrange, **VAL** returns the element with the position **e** in the original enumeration type, not the subrange. A checked runtime error results when the value of **e** is out of range.

If **n** is an integer, then **ORD(n) = VAL(INTEGER, n) = n**. It is a static error to apply **ORD** and **VAL** to **UNSIGNED**.

```
FIRST (T: OrdinalOrRealType): T;
LAST  (T: OrdinalOrRealType): T;
```

FIRST returns the smallest value of the type **T**, which should be an ordinal type or **REAL** or **LONGREAL**. **LAST** returns the largest value.

If **T** is an enumeration, **FIRST** returns the leftmost enumeration constant and **LAST** returns the rightmost. If **T** is a subrange, the result is the first or last value in the subrange, not the original enumeration. It is a static error to call **FIRST** or **LAST** on an empty enumeration or subrange.

FIRST(INTEGER), **FIRST(UNSIGNED)**, **FIRST(REAL)**, or **FIRST(LONGREAL)** yields the smallest possible value allowed by the implementation; **LAST(INTEGER)**, **LAST(UNSIGNED)**, **LAST(REAL)**, or **LAST(LONGREAL)** yields the largest.

```
LOW (a: Array)           : Ordinal;
    (AT: FixedArrayType) : Ordinal;

HIGH (a: Array)           : Ordinal;
     (AT: FixedArrayType) : Ordinal;
```

LOW returns the smallest index of array **a** or the smallest index of the fixed array type **AT**. **HIGH** returns the largest index. If the array or array type is empty, **LOW** returns 0 and **HIGH** returns -1.

If **a** is a fixed array, the result is a constant of the index type of the array. If **a** is an open array, the result is an **INTEGER**.

```
NUMBER (ot: OrdinalType)    : CARDINAL;
       (AT: FixedArrayType) : CARDINAL;
       (a: Array)           : CARDINAL;
```

NUMBER returns the number of elements in the ordinal type **ot**, the array **a**, or the fixed array type **AT**. In the first and second cases, the result is always a constant; in the third case the result is a constant if the syntactic type of **a** is a fixed array type. It is a checked runtime error for the type or array to have more than **LAST (CARDINAL)** elements.

```
NARROW (x: Value; T: Type): T;
```

Checks that the value of **x** is a member of **T** and returns the value. A checked runtime error occurs otherwise (with the exception described below). It is a static error if **T** is not a ref or pointer type or the syntactic type of **x** is not assignable to **T**.

In safe modules, it is a static error if if the syntactic type of **x** is **ADDRESS** and **T** is a pointer type (6.5, page 45). In unsafe modules, it is an unchecked runtime error if **x** does not represent a valid pointer value.

```
LOOPHOLE (x: Value; T: Type): T;
```

Returns **x**'s bit pattern interpreted as a value of type **T**. It is a static error if **BITSIZE (x)** is not equal to **BITSIZE (T)** or if **T** or the type of **x** is an open array.

In safe modules, it is a static error if **T** is not representation complete (2.5, page 13, 6.5, page 45). In unsafe modules, it is an unchecked error if **x**'s bit pattern is not a legal value of **T**.

```
TYPECODE (x: RefOrRefany): CARDINAL
         (x: Type)       : CARDINAL
```

If the syntactic type of **x** is a ref type or **REFANY**, returns the type code associated with the referent of **x**. If **x** is a ref type, returns the type code assigned to it. See 2.2.5, page 10.

```
BITSIZE  (x: Any) : CARDINAL;
         (T: Type): CARDINAL;
BYTESIZE (x: Any) : CARDINAL;
         (T: Type): CARDINAL;
```

BITSIZE returns the number of bits used by the implementation to represent the value **x** or a value of type **T**. **BYTESIZE** returns the number of 8-bit bytes.

It is a static error if **x** is an open array or **T** an open-array type.

```
ADDR  (VAR IN x: Anything): ADDRESS;
```

Returns the address of the variable designated by **x**; the exact meaning is

implementation dependent.

3.2.8. Selectors

e^

e^ is the variable addressed by **e**. The syntactic type of **e** must be a concrete ref or pointer type.

d[e]

d[e] is element **e - FIRST(d)** of **d**. **d** can be any array-valued expression. The syntactic type of **e** must be assignable to the index type of **d**. The syntactic type of **d[e]** is the element type of **d**. Indexing out of bounds is a checked runtime error.

d.f

If **d** is a record-valued expression, **d.f** denotes the field named **f**.

If **d** names an imported interface, then **d.f** denotes the entity named **f** in the interface **d**.

3.3. Constant Expressions

Constant expressions are a subset of the general class of expressions, restricted by the requirement that it be possible to evaluate the expression at compile time. The following operations are legal in constant expressions:

```
+  -   *  /
=  #   <  >  <=  >=
.
```

ABS	**BITSHIFTRIGHT**	**IN**	**OR**
AND	**BITSIZE**	**LAST**	**ORD**
BITAND	**BITXOR**	**LONGFLOAT**	**ROUND**
BITEXTRACT	**BYTESIZE**	**LOW**	**TRUNC**
BITINSERT	**CEILING**	**MAX**	**VAL**
BITNOT	**DIV**	**MIN**	
BITOR	**FIRST**	**MOD**	
BITROTATELEFT	**FLOAT**	**NARROW**	
BITROTATERIGHT	**FLOOR**	**NOT**	
BITSHIFTLEFT	**HIGH**	**NUMBER**	

A variable can appear in a constant expression only as an argument to **FIRST**, **LAST**, **NUMBER**, **BITSIZE**, and **BYTESIZE**, and such a variable must not be an open array.

The only procedure-valued or reference-valued constant expression is **NIL**.

4. Declarations

Every user-defined identifier must be introduced by a declaration, which binds the identifier to a constant, type, variable, exception, or procedure. Declarations appear in a *block* of the form:

```
CONST ... ;
TYPE ... ;
VAR ... ;
EXCEPTION ... ;
PROCEDURE ... ;
BEGIN
    Statements
END
```

A block can appear as a statement or as the body of a module or procedure. The effect of a name binding extends from the beginning of the block to the end of the block.

In general, a name can be declared at most once in any block, though a name can be redeclared in nested blocks. The meaning of a name at any point is determined by the smallest enclosing block in which the name is defined. The order of declarations within a block does not matter.

Mutually referential declarations are allowed only for procedures and types. Mutually referential procedure declarations are always legal. Mutually referential type declarations are legal if every cyclic path of types contains at least one **REF**, **POINTER**, or **PROCEDURE** constructor. So this is legal:

```
TYPE T1 = RECORD t2: T2 END;
     T2 = REF T1;
```

And this is not:

```
TYPE T1 = RECORD t2: T2 END;
     T2 = RECORD t1: T1 END;
```

Modula-2+ also has a set of identifiers that are declared in a scope around all modules. For a complete list, see 7, page 47.

4.1. Constants

If `ConstExpr` is a constant expression then:

```
CONST id = ConstExpr
```

declares a new constant `id` whose syntactic type is that of `ConstExpr`.

Declaring an enumeration type also declares the constants that are elements of the type. The scope of the enumeration constants is the same as the scope of the enumeration itself.

A constant `C` may be trivially redeclared in the same scope:

```
CONST C = U;
CONST C = V;
```

only if `U` and `V` denote the same constant value.

4.2. Types

If U is any expression denoting a type, a type declaration of the form:

```
TYPE T = U
```

declares T to be the type denoted by U. If U denotes an enumerated type, then all the identifiers naming constants of the enumeration are also implicitly declared in the same scope as T.

A type T may be trivially redeclared in the same scope:

```
TYPE T = U;
TYPE T = V;
```

only if U and V denote the same type.

An opaque ref type is declared by:

```
TYPE T = REF
```

An opaque word type is declared by:

```
TYPE T
```

A concrete type V can be identified with an already-declared opaque type T by:

```
TYPE T = V
```

T may be a qualified identifier. If T is an opaque ref type, then V must be a ref type. If T is an opaque word type, then V must be a non-ref type, **BITSIZE(V)** must be equal to **BITSIZE(ADDRESS)**, and in safe modules, V must be representation complete.

The scope of the type identification is the same as the scope that a declaration would have in the same position. Importing an interface implicitly imports all of its type identifications (see 6.2, page 43 and 6.3, page 44). When T is visible in a scope but no concrete type has been identified with T in that scope, T is said to be *opaque* in the scope.

Two already-declared opaque ref types U and V can be identified:

```
TYPE U = V
```

Within the scope of the the identification, U and V are considered to be the same type.

Opaque type declarations and type identifications can appear in either interfaces or implementations. It is a static error for a program to identify distinct concrete ref types, either directly or indirectly through intermediate opaque types.

Variables with an opaque syntactic type T cannot be allocated with **NEW**, deallocated with **DISPOSE**, or dereferenced.

4.3. Variables

If `Type` is any type, then:

```
VAR id: Type
```

declares a variable of type `Type` and binds `id` to that variable.

If `QualID` is a qualified identifier, a variable declaration can also take the form:

```
VAR id: Type = QualID
```

which creates `id` as an alias for `QualID`. The type of `QualID` must be the same as `Type`. Such aliases can be declared in interfaces only.

In either form, when a set of identifiers share the same type, `id` can be a list of identifiers separated by commas:

```
VAR v1, ..., vn : type;
```

This is shorthand for:

```
VAR v1 : type;
  ...
VAR vn : type;
```

A name `id` may be trivially redeclared as a variable in a scope only if the two declarations bind `id` to the same variable through the use of aliases:

```
VAR id: Type = qualid;
...
VAR id: Type = qualid;
```

The initial value of a variable of type `T` is guarranteed to be a legal value of `T`. Refs and pointers are initalized to `NIL`.

4.4. Exceptions

If `TypeID` is a qualified identifier denoting any type other than an open array type, then:

```
EXCEPTION idList ( TypeID )
```

declares the named exceptions. `idList` is the name of an exception or a list of names separated by commas; `TypeID` is an optional argument type for the exception.

An exception declaration can also take the form:

```
EXCEPTION idList ( TypeID ) = QualID
```

which creates each `id` in `idList` as an alias for `QualID`. Either both sides must have an argument and the argument types must be the same, or neither side must have an argument. Such aliases can be declared in interfaces only.

4.5. Procedures

A procedure declaration has the form:

```
PROCEDURE ident (
  mode1 name1: type1 := default1
  ...
  moden namen: typen := defaultn ) : R
RAISES S;

Declarations;
BEGIN
    Statement
END ident.
```

The semantics of a procedure signature are exactly like that for a procedure type (2.2.8, page 11). The scope of a procedure begins after the procedure name: the procedure name is in the enclosing block, but the parameter names are in the procedure block.

The $\texttt{name}_i$ can be a list of identifiers separated by commas:

```
mode name1, ..., namen: type := default;
```

This is shorthand for:

```
mode name1: type := default;
...
mode namen: type := default;
```

A procedure declaration can also take the form:

```
PROCEDURE ident Signature = QualID;
```

which creates `ident` as an alias for the procedure `QualID`. The types of the left and right hand sides must be assignable. Such aliases can be declared in interfaces only.

The keyword **INLINE** can optionally precede any procedure declaration; it identifies a procedure that is intended to be expanded at the point of call. **INLINE** does not affect the language semantics; it is a hint for the implementation.

A *procedure definition* is a procedure signature without the body.

The raises set `S` of a procedure is equivalent to a **TRY PASSING** (5.6, page 31) around the procedure body `Statement`:

```
TRY Statement PASSING S END
```

A name `id` may be trivially redeclared as a procedure in a scope only if the two declarations bind `id` to the same procedure through the use of aliases:

```
PROCEDURE P() = Q;
...
PROCEDURE P() = Q;
```

5. Statements

This chapter defines the Modula-2+ statements. Executing a statement produces a computation that either:

- halts (normal outcome)
- raises an exception (exceptional outcome)
- loops forever or causes a checked runtime error (looping/error outcome)

Each exceptional outcome is tagged with an exception `e`, together with an argument value if `e` takes an argument. Raising an exception has no effect on the values of variables.

In addition to the explicitly declared exceptions, we use two additional kinds of exceptions. The return-exception is used to define the semantics of **RETURN**, and exit-exceptions are used to define the semantics of **EXIT**.

5.1. Assignment

If `u` and `t` are expressions with syntactic type `U` and `T`, then `u` is *assignable* to `t` if:

> `t` is a designator;
>
> `U` is assignable to `T` (2.4, page 12), and in safe modules, if `T` is a pointer type, `U` is not **ADDRESS**.
>
> `u` is a member of `T`, `u` is not a local procedure, and if `u` and `t` denote arrays, the arrays have the same number of elements.

The last requirement generally requires a runtime check; in unsafe modules, it is an unchecked runtime error if `T` is a pointer type and `u` is not a member of `T` (6.5, page 45). If the type of `t` is integer or subrange of integer and the type of `u` is unsigned or subrange of unsigned, or vice versa, then the value of `u` will be loopholed into the type of `t` without any runtime checking.

An expression `u` is assignable to a type `T` if `u` is assignable to a variable of type `T`.

The assignment statement has the form:

```
t := u
```

where the expression `u` is assignable to the expression `t`. The statement changes the value of the variable denoted by `t` to the value of the expression `u`.

The order of evaluation of `t` and `u` is undefined, but `u` will be evaluated before the variable denoted by `t` is updated.

5.2. Sequential composition

A statement of the form:

```
S1; S2
```

executes S_1, and then if the outcome is normal, executes S_2. If the outcome of S_1 is an exception, then S_2 is ignored. If the outcome of either S_1 or S_2 is an exception, then the outcome of the composition is that exception.

5.3. RAISE

A **RAISE** statement without an argument has the form:

```
RAISE(e)
```

where **e** is an exception. The statement causes an exceptional outcome tagged with **e**.

A **RAISE** statement with an argument has the form:

```
RAISE(e, a)
```

where **e** is an exception and **a** is any expression that is assignable to **e**'s argument type. The statement causes an exceptional outcome tagged with **e** and **a**.

It is a checked runtime error if the call to **RAISE** is not dynamically contained in a **TRY EXCEPT** scope that will handle **e**. A **TRY EXCEPT** scope "handles **e**" if **e** or **ELSE** is listed in its handlers and if there are no intervening **TRY EXCEPT** scopes that handle **e** and if every intervening **TRY PASSING** scope or procedure **RAISES** clause includes **e**.

5.4. TRY EXCEPT

A **TRY-EXCEPT** statement has the form:

```
TRY
  Body
EXCEPT
| id1 (v1) : Handler1
...
| idn (vn) : Handlern
| ELSE                Handler0
END
```

where **Body** and each **Handler** are statements, each **id** is a qualified identifier that names an exception, and each **v** is an identifier. The "**ELSE Handler**$_0$" is optional. The text between "**EXCEPT**" and "**END**" is called the *except clause*. It is a static error for an exception to occur more than once in the list of **id**s.

The statement executes **Body**. If the outcome is normal, the except clause is ignored.

If **Body** raises any listed exception $\mathbf{id}_i$, then $\mathbf{Handler}_i$ is executed. If **Body** raises any unlisted exception and $\mathbf{Handler}_0$ is present, then it is executed. In either case, the outcome of the **TRY** statement is the outcome of the **Handler**. If **Body** raises an unlisted exception and $\mathbf{Handler}_0$ is absent, then the outcome of the **TRY** statement is the exception raised by **Body**.

Each $\mathbf{v}_i$ must be a variable whose syntactic type is the same as the argument type of the exception $\mathbf{id}_i$. When an exception tagged with $(\mathbf{id}_i, \mathbf{a})$ is handled, **a** is assigned to $\mathbf{v}_i$ before $\mathbf{Handler}_i$ is executed. The variable $\mathbf{v}_i$ may be omitted even if

the exception $\texttt{id}_i$ takes an argument.

When a series of exceptions share the same handler, $\texttt{id}_i$ can be a list of qualified identifiers separated by commas. Such a list is shorthand for an expansion in which the rest of the handler is repeated for each qualified identifier in the list. That is:

```
id1, ..., idn (v): Handler
```

is shorthand for:

```
id1 (v): Handler;
  ...
idn (v): Handler;
```

5.5. TRY FINALLY

A statement of the form:

```
TRY
  Statement1
FINALLY
  Statement2
END
```

executes $\texttt{Statement}_1$ and then $\texttt{Statement}_2$. If the outcome of $\texttt{Statement}_1$ is normal, the **TRY** statement is equivalent to $\texttt{Statement}_1$; $\texttt{Statement}_2$. If $\texttt{Statement}_1$ raises an exception and $\texttt{Statement}_2$ does not, the exception from $\texttt{Statement}_1$ is re-raised after $\texttt{Statement}_2$ is executed. If both raise exceptions, the outcome of the **TRY** is the exception from $\texttt{Statement}_2$.

5.6. TRY PASSING

A statement of the form:

```
TRY
  Statement
PASSING {id1, ..., idn}
END
```

executes **Statement**. If the outcome is normal, the outcome of the **TRY** is normal. If the outcome is one of the exceptions $\texttt{id}_i$ or a return or loop-exit exception, then the outcome of the **TRY** is that exception. It is a checked runtime error if the outcome is any other exception.

5.7. LOOP

A statement of the form:

```
LOOP
  Statement
END
```

repeatedly executes **Statement** until it raises the exit-exception or terminates with some other exception. That is, it is equivalent to:

```
TRY
  Statement; Statement; Statement ...
EXCEPT
  exit-exception_loop:
END
```

Every loop has its own unique exit-exception.

5.8. EXIT

A statement of the form:

```
EXIT
```

raises the **exit-exception**$_{loop}$ of the innermost, lexically enclosing **LOOP**. It is a static error if there is no such enclosing loop.

5.9. WHILE

If **Expr** is an expression of type **BOOLEAN**, a statement of the form:

```
WHILE Expr DO
  Statement
END
```

repeatedly evaluates **Expr** and executes **Statement** as long as **Expr** yields the value **TRUE**. If **Expr** is initially **FALSE**, **Statement** never executes. The above **WHILE** loop is roughly equivalent to:

```
LOOP
  IF NOT Expr THEN EXIT END;
  Statement
END
```

("roughly" because **EXIT** cannot be used to exit a **WHILE** loop, as this expansion implies.)

5.10. REPEAT

If **Expr** is an expression of type **BOOLEAN**, a statement of the form:

```
REPEAT
  Statement
UNTIL Expr
```

repeatedly executes **Statement** and evaluates **Expr** until **Expr** has the value **TRUE**. **Statement** always executes at least once. The above loop is roughly equivalent to:

```
LOOP
  Statement;
  IF Expr THEN EXIT END
END
```

("roughly" because **EXIT** cannot be used to exit a **REPEAT** loop, as this expansion implies.)

5.11. FOR

A statement of the form:

```
FOR id := first TO last BY step DO
  Statement
END
```

steps the variable **id** through the sequence of values **first**, **first+step**, **first+2*step**, . . . , stopping when the value of **id** passes **last**. The **Statement** executes once for each value of **id**. **first** and **last** are evaluated once before the loop is entered; if the sequence of values is empty, the **Statement** never executes.

The identifier **id** must have ordinal type. The type of the expressions **first** and **last** must be assignable to **id**.

step is an optional **INTEGER** constant expression that specifies an increment or decrement; the default is 1. If **step** is negative, the loop iterates downward.

If **T** is the syntactic type of **id**, the above **FOR** loop is equivalent to:

```
VAR iTemp, lTemp: INTEGER;

iTemp := ORD(first);
lTemp := ORD(last);
WHILE ((step >= 0) AND (iTemp <= lTemp)) OR
      ((step <  0) AND (iTemp >= lTemp))
DO
  id := VAL(T, iTemp);
  Statement;
  iTemp := iTemp + step;
END;
```

5.12. IF

A statement of the form:

```
IF    Expression1 THEN Statement1
ELSIF Expression2 THEN Statement2
   ...
ELSIF Expressionn THEN Statementn
ELSE Statement0
END
```

executes at most one of the **Statement**s based on the values of the **Expression**s, which must be of type **BOOLEAN**. The **ELSIF** and **ELSE** clauses are optional. The expressions are evaluated in order until one yields the value **TRUE**; when **Expression**$_i$ yields **TRUE**, **Statement**$_i$ is executed. If none of the expressions evaluates to **TRUE**, **Statement**$_0$ is executed, if present.

5.13. CASE

A **CASE** statement has the form:

```
CASE Expr OF
| S1 : Statement1
| ...
| Sn : Statementn
ELSE Statement0
END
```

where **Expr** is any expression denoting an ordinal, and each **S** is a list of constant expressions or ranges denoted by "$e_1 .. e_2$", where e_1 and e_2 are constant expressions. Each **S** represents the set containing the listed constants and ranges. It is a static error if any two **S**'s overlap or if the type of any of the constant expressions is not assignable to the type of **Expr**.

The "**ELSE Statement**$_0$" is optional.

The statement evaluates **Expr**. If the resulting value is in any S_i, then **Statement**$_i$ is executed. If the value is in no S_i and **Statement**$_0$ is present, then it is executed. If the value is in no S_i and **Statement**$_0$ is absent, a checked runtime error occurs.

5.14. TYPECASE

A **TYPECASE** statement has the form:

```
TYPECASE Expr OF
| T1 (v1) : Statement1
  ...
| Tn (vn) : Statementn
ELSE Statement0
END
```

where **Expr** is an expression whose syntactic type is a ref type or **REFANY**, each T_i denotes a ref type, and each v_i is an identifier whose type is T_i. It is a static error if these restrictions are not met.

The "**ELSE Statement**$_0$" and each "**(v)**" are optional.

The statement evaluates **Expr**. If the resulting value is a member of the value set of any listed type T_i, then **Statement**$_i$ is executed, for the minimum such **i**; if v_i is present, it is assigned the value of **Expr** before **Statement**$_i$ is executed. If the value is a member of no listed type and **Statement**$_0$ is present, then it is executed. If the value is a member of no listed type and **Statement**$_0$ is absent, a checked runtime error occurs.

If a series of branches share the same statement and none of them specify a variable, then T_i can be a list of type expressions separated by commas. Such a list is shorthand for an expansion in which the rest of the branch is repeated for each type expression in the list. That is:

```
T1, ..., Tn : Statement
```

is shorthand for:

```
| T1 : Statement
| ...
| Tn : Statement
```

5.15. Procedure call

If P is an expression whose value is a procedure, a statement of the form:

P($value_1$, ..., $value_n$)

denotes execution of the named procedure with the actual parameters $value_1$, ..., $value_n$ substituted for the formal parameters. If P is a function procedure, the value returned by the call is discarded.

If there are fewer actuals than formals and the remaining formals have defaults, the default values are supplied. It is a static error to not supply an actual for a formal that doesn't have a default.

There are four modes for parameter passing: value, **VAR**, **VAR IN**, and **VAR OUT**.

Within the called procedure, a by-value parameter is equivalent to a local variable that is assigned the value of the actual argument at the point of call; see the assignment statement, 5.1, page 29.

All forms of **VAR** parameters mean "by reference". A **VAR** parameter designates a variable that is selected when the procedure is called; every operation the procedure body performs on the formal parameter is performed on the actual variable. **VAR IN** is an input-only specialization of **VAR**; the callee cannot assign to a **VAR IN** parameter or pass it as a **VAR** or **VAR OUT** parameter. **VAR OUT** is an output-only specialization of **VAR**; the initial value of the formal is undefined.

The actual passed to a by-value or **VAR IN** parameter can be any expression that is assignable to the type of the formal; see 5.1, page 29. If the syntactic type of a **VAR IN** actual parameter is not the same type as the formal, the actual is assigned to a temporary variable and that variable is passed by reference to the formal.

With the following exceptions, the actual passed to a **VAR** or **VAR OUT** parameter must be a designator whose type is the same as the formal. An array actual can be passed to a **VAR** or **VAR OUT** open array formal if the element types of the arrays are the same. Any actual can be passed to a **VAR** or **VAR OUT BYTE**, **WORD**, **ARRAY OF BYTE**, or **ARRAY OF WORD** formal if the actual type is assignable to the formal and, in safe modules, if the actual type is representation complete (6.5, page 45).

In some implementations it may not be possible to take the address of expressions whose types are **BITS FOR** types; implementations are thus free to forbid some or all **BITS FOR** types from being passed by any form of **VAR**.

Here is a table that classifies the four modes according to the kind of actual that can be passed in (expression or designator), whether the value of the formal is initialized to the value of the actual, whether the caller can update the formal, and whether the updates are visible to the caller:

	value	VAR IN	VAR	VAR OUT
actual	exp	exp	des	des
initialized	yes	yes	yes	no
updatable	yes	no	yes	yes
updates visible	no	N/A	yes	yes

5.16. RETURN

If `Expr` is an expression, a statement of the form:

```
RETURN e
```

terminates the current activation of the procedure in which it occurs by raising the return-exception. `e` denotes the returned value. It is a static error to include a return value in a proper procedure or to omit one in a function procedure. `e` must be assignable to the return-value type of the procedure.

A call on a proper procedure with body `B` is equivalent to:

```
TRY
  B
EXCEPT
  return-exception:
END
```

A call on a function procedure with body `B` is equivalent to:

```
TRY
  B;
  "error: no returned value"
EXCEPT
  return-exception (v): "the result becomes v"
END
```

5.17. WITH

If `e` is an expression of record-type `T`, a statement of the form:

```
WITH e DO
   Statement
END
```

allows a record field `f` of the value of `e` to be referenced within the statement simply as `f` instead of `e.f`. That is, `WITH` provides a new nested scope containing the record-field names as aliases for the fields of the record variable `e`. The aliases are designators if `e` is a designator. The value of `e` is computed once, at entry to the `WITH` statement.

5.18. LOCK

If `mu` is a designator whose type is Thread.Mutex (1.5, page 5), a statement of the form:

```
LOCK mu DO
  Statement
END
```

executes `Statement` while holding the mutex. This statement isroughly equivalent to:

```
Thread.Acquire(mu);
TRY
  Statement;
FINALLY
  Thread.Release(mu);
END
```

except that `mu` is evaluated only once on entry to `LOCK`.

5.19. NEW

If v is a designator whose syntactic type is a concrete pointer or ref type, a statement of the form:

```
NEW(v)
```

sets v to the address of a newly-allocated variable of v's referent type.

If v's referent type is an open array, then NEW requires an additional argument:

```
NEW(v, n)
```

n must be a CARDINAL value, specifying the number of elements of the array.

If v's referent type is a record containing a variant field, then an optional second argument is allowed but not required:

```
NEW(v, t)
```

t is an expression specifying the variant tag for the first variant field of the record. The type of t must be assignable to the type of the variant tag. There is no way to specify tags for other than the first variant field or for nested variant records. Nor is there any way to specify the tags of an open array of variant records.

5.20. DISPOSE

If v is a designator whose type is a concrete pointer or ref type or REFANY, a statement of the form:

```
DISPOSE(v)
```

sets v to NIL. If v is a concrete pointer, the storage for its referent is also freed; freeing storage to which active references remain is an unchecked runtime error, and thus DISPOSE applied to pointer types is prohibited in safe modules (6.5, page 45).

5.21. INC and DEC

If x is a designator whose type is ordinal, pointer, or ADDRESS, statements of the form:

```
INC(x, n)
DEC(x, n)
```

increment and decrement x by n, respectively. n is an optional parameter of type INTEGER; the default is 1.

If x's type is numeric, pointer, or ADDRESS, then

```
INC(x,n)  =     x := x + n
DEC(x,n)  =     x := x - n
```

except that the designator x is only evaluated once. Applying INC or DEC to pointers is not allowed in safe modules (6.5, page 45).

If the type of x is an enumeration type T, then:

```
INC(x,n) =    x := VAL(T, ORD(x) + n)
DEC(x,n) =    x := VAL(T, ORD(x) - n)
```

except that the designator x is only evaluated once.

5.22. INCL and EXCL

If **s** is a designator that denotes a set, statements of the form:

```
INCL(s, e)
EXCL(s, e)
```

add an element **e** to a set **s** or remove an element **e** from a set **s**, respectively. The type of **e** must be assignable to the element type of **s**.

5.23. PRINTF

```
PRINTF(
    wr: Writer;
    format: Text.T;
    ...arguments... )
    RAISES {Wr.Failure, Wr.Error, Wr.Alerted};
```

PRINTF prints and formats its arguments to the writer **wr**, which is a **Wr.T** or **WrV.T**. The **format** must be a constant text, and it determines the number and types of the arguments that follow.

The format text consists of literal text, which is written to **wr** exactly as it appears, and format items, which specify output conversions to be done.

A format item consists of a "%" sign, optionally followed by formatting information (described below), followed by the format character, which specifies the type of output conversion to take place. For every format item in the format text, there must be a corresponding argument in the arguments to **PRINTF** following the format text. The order of format items is the same as the order of their corresponding arguments. The argument must be assignable to the format item's type. Its value is formatted as appropriate and written out to **wr**. The format items, their types, and their functions are as follows:

%d	**INTEGER**; prints as a signed decimal number.
%u	**INTEGER**; prints as an unsigned decimal number.
%o	**INTEGER**; prints as an unsigned octal number with no leading "0" or trailing "B".
%x	**INTEGER**; prints as an unsigned hexadecimal number with no leading "0x" or trailing "H".
%e	**REAL**; prints in scientific notation.
%f	**REAL**; prints in fixed point notation.
%g	**REAL**; prints in fixed point or scientific notation, whichever is shorter.
%le %lf %lg	**LONGREAL**; like %e, %f, and %g.
%c	**CHAR**; prints the character unformatted.
%s	**ARRAY OF CHAR**; prints the characters unformatted. Printing will stop at the end of the array, upon reaching a NUL character (a zero byte), or when the maximum width is reached (see below).
%t	**Text.T**; prints the characters of the text unformatted.
%%	prints a '%' character.

The full syntax for a format item is:

```
%[-][0][<width>][.<precision>][#]<format character>
```

The **precision** can be specified only for formats **%e**, **%f**, **%g**, **%le**, **%lf**, **%lg**, **%ls**. The # can be specified only for formats **%o**, **%x**, **%e**, **%f**, **%g**, **%le**, **%lf**, **%lg**.

The **width** and **precision** are unsigned decimal integers. **width**, if given, specifies the minimum width of the field to be printed. This is designed to make printing of aligned tables easy. If the actual item to be printed comes out larger than the field width, the field will be larger, but if it comes out smaller, the field will be padded. The default is to pad on the left with blanks. The "-", if given, specifies left justification; the field will be padded on the right instead of on the left. A leading zero on the **width** specifies zero fill; the field will be padded with zeroes instead of with blanks. Zero fill only works on the numeric format items. Using left justification and zero fill together is inadvisable; it puts extra zeroes after the numbers, making them look very large.

In the formats that print floating-point numbers, the **precision** argument specifies the number of digits to be printed to the right of the decimal point. The default is six digits. Note: the maximum supported width for numerical fields is 40. In the formats that print variable-length strings (**%s** and **%t**), the **precision** argument specifies the maximum number of characters that will be printed. If the string or text is longer, only the first **precision** characters of it will be printed.

The "#", if present, specifies an alternate form for printing. This applies only to a few format items:

%o prints a leading "0", the C convention for octal numbers.

%x prints a leading "0x", the C convention for hexadecimal numbers.

floating point formats -- prints a trailing decimal point, regardless of whether one is needed.

The **width** and **precision** may be given as a "*", instead of a number. This means variable width or variable precision. The actual value for **width** or **precision** is determined at runtime by an extra argument to **PRINTF**. These arguments, which are of type **INTEGER**, come before the argument for the format item itself. An example:

```
PRINTF(wr, "%*.*f", width, precision, real);
```

PRINTF raises **Wr.Failure** if something goes wrong with the writer during output (string full, lost connection, broken pipe, etc.) It raises **Wr.Error(Closed)** if applied to a closed writer.

PRINTF is intended to be fully compatible with the standard C function **printf**(3).

5.24. SCANF

```
PROCEDURE SCANF(
    rd: Reader;
    format: Text.T;
    ...arguments... )
    RAISES {Rd.EndOfFile, Rd.ScanFailed, Rd.Failure, Rd.Error,
Rd.Alerted};
```

SCANF converts printed representations of values taken from an input stream into Modula-2+ values and stores the values into the arguments. The input stream **rd** must be an **Rd.T** or **RdV.T**. The **format** must be a constant text, and it determines the number and types of the arguments that follow.

The format string consists of:

whitespace, which matches optional whitespace from the reader **rd**,

literal nonwhite characters, which matches the identical nonwhite characters from **rd**. **SCANF** raises **Rd.ScanFailed** if the two do no match, and

format items, which specify input conversions to be done.

Whitespace means a run of space, tab, and newline characters.

A format item consists of a "%", optionally followed by width and skip specifications (see below), followed by the format character, which specifies the type of input conversion to take place. For every format item in the format string, there must be a corresponding argument in the arguments to **SCANF** following the format string. The order of format items is the same as the order of their corresponding arguments. The argument is taken as a **VAR OUT** parameter, and is assigned a value obtained from the input stream. The format items and their functions are as follows:

%d	decimal input expected, argument should be an **INTEGER** variable.
%o	octal input expected, argument should be an **INTEGER** variable with no trailing "**B**".
%x	hexadecimal input expected, argument should be an **INTEGER** variable with no trailing "**H**" or leading "**0x**".
%e %f	a floating-point number is expected, argument should be a **REAL** variable. The number may optionally include a decimal point and/or the letter "**E**" or "**e**" followed by an exponent.
%le %lf	like **%e** or **%f**, except that the argument should be a **LONGREAL**.
%c	will take one character from the input, argument should be a **CHAR** variable.
%<number>c	will take **<number>** characters from the input, argument should be an **ARRAY OF CHAR**, at least as big as **<number>**.
%t	will take a string, delimited by whitespace or end of file, from the input. A text containing the scanned characters will be created and assigned to the variable, which should be a **Text.T**.
%s	will take a string, delimited by whitespace or end of file, from the input. Argument should be an **ARRAY OF CHAR**. If the

array fills up before the string is all scanned, scanning will stop in mid-string. If the array is not filled up, a NUL terminator (a zero byte) will be placed in the array following the string.

`%[<set of characters>]`
: scans a field defined by the set of characters. If the character following "`[`" is not "`^`", **SCANF** reads characters until a character `c` not present between the "`[`" and "`]`" is seen; otherwise **SCANF** reads characters until a character `c` present between the "`^`" and "`]`" is seen. The character `c` is put back into the reader using **Rd.UngetChar**. If the argument is a **Text.T** variable, a **Text.T** is created containing the characters prior to `c`, and this **Text.T** is assigned to the variable. If the argument is an **ARRAY OF CHAR variable**, the zero or more characters prior to `c` are copied sequentially into the array, followed by NUL if there is room.

`%%`
: scans a literal '`%`' character, raising **Rd.ScanFailed** if the next input character doesn't match.

For all format items except `%c` and %[], **SCANF** will automatically skip over white space (including newlines) in search of the beginning of the field. (Hence whitespace is significant only before `%c` or `%[]`, or at the end of the format.) Upon finding a non-whitespace character, it begins to do the specified input conversion. If it doesn't find what it was looking for, it raises **Rd.ScanFailed** (e.g. a letter where a decimal number was expected). Scanning of that field continues until a character is encountered that cannot be interpreted as part of that field, or until the specified field width (see below) is reached. Example: a format item "`%d`", encountering "27a95" in the input, will deliver 27 as the number, and leave the reader positioned at "a". Note: `%[]` items will match the empty string without complaint, and will never raise **Rd.ScanFailed**.

Note that "bare" numeric fields are expected: `%x` takes a string of hexadecimal digits, plain and simple. No leading "`0`" is ever required, and a trailing "`H`" is not taken as part of the field. Nor is the C convention of a leading "`0x`" recognized. Likewise, `%o` does not include a trailing "`B`" in the field; it only takes a string of digits 0-7.

SCANF raises **Rd.EndOfFile** if it reaches end of file while skipping whitespace looking for a field, or while handling a `%c` format item. It raises **Rd.Error**(Closed) if applied to a closed reader. It raises **Rd.Failure** if something goes wrong with the reader while **SCANF** is reading from it.

The full syntax for a format item is:

```
%[*][<width>]<format character>
```

The "`*`", if present, means that a field of this description is to be skipped. In this case, no corresponding argument should appear in the arguments to **SCANF**. The width, if present, is the maximum width of the field to be scanned for this item. A shorter field will be accepted, but scanning of this item will stop after width characters, regardless of what comes next. For example, a format item "`%2d`", encountering "2347" in the input, will deliver 23 as the number and leave the reader positioned at "4".

SCANF is intended to be fully compatible with the standard C function **scanf**(3).

5.25. COPY and ZERO

```
COPY( from, to: ADDRESS; n: CARDINAL )
```

Copies n bytes from `from` to `to`. `COPY` may not be used in safe modules (6.5, page 45).

```
ZERO( to: ADDRESS; n: CARDINAL )
```

Stores zero into the n bytes starting at `to`. `ZERO` may not be used in safe modules (6.5, page 45).

5.26. ASSERT

If `condition` is an expression of type `BOOLEAN`, a statement of the form:

```
ASSERT(condition, message)
```

tests the value of `condition` and causes a checked runtime error if it is false. `message` is an optional argument of type `Text.T` that can be used to help identify the particular error.

5.27. HALT

If n is an expression whose type is `INTEGER`, then

```
HALT(n)
```

halts the program, passing the value of n as the exit code to the operating system. `n` is optional and defaults to `0`.

6. Interfaces and implementations

A *module* is a collection of declarations and statements used to introduce hierarchical structure into a program. There are two kinds of modules, *interfaces* and *implementations*. An interface contains only declarations; an implementation contains both declarations and statements.

6.1. Interfaces

An interface has the form:

```
DEFINITION MODULE ident ;
Imports
Declarations
END ident .
```

where **Declarations** are constant, variable, type, and exception declarations, and procedure definitions. The purpose of an interface is to define symbols for other modules to "import." An interface contains only constant declarations, static typing information, and inline procedures; it cannot contain non-inline procedure bodies or any other executable statements.

6.2. Implementations

An implementation module has the form:

```
MODULE ident IMPLEMENTS idList ;
Import;
Declarations
BEGIN
  Statement
END ident .
```

where **IdList** is a list of interface names, separated by commas, **Declarations** are declarations of constants, variables, types, exceptions, and procedures, and **Statement** is the "main body" of the module.

If an implementation module **M** declares a procedure whose name matches a procedure in one of the interfaces that **M** implements, that procedure is *exported* to the interface and thereby made available to clients of the interface. The **IMPLEMENTS** clause can be omitted if the module implements no interfaces. An implementation that implements no interfaces is a main program.

The declaration of an exported procedure need only be assignable to the corresponding procedure declaration in the interface. The two can thus differ in parameter names and results. Within the implementation, if the declarations of a procedure differ, the declaration in the implementation takes precedence over that in the interface.

The body of an exported inline procedure must appear in the interface, not in the implementation.

All declared or imported names in an interface **D**, including constants of enumerations but excluding imported interfaces, are available without qualification in the scope of a module that implements **D**. All type identifications (4.2, page 26) in **D** are also visible in the scope of the implementing module.

An implementation module can also have the form:

```
IMPLEMENTATION MODULE ident ;
Import
Declarations
BEGIN
  Statement
END ident .
```

which is a shorthand way to specify that the module implements only the interface **ident**.

It is a static error for an implementation module to implement two interfaces declaring the same name. There can be at most one implementation module for a given interface. To get the effect of multi-module implementations, use a structure of the form:

```
DEFINITION MODULE Umbrella;
IMPORT Part1, Part2, Text;
TYPE   T = Part1.T;

PROCEDURE Create(name: Text.T): T = Part1.Create;
PROCEDURE Print(t: T) = Part2.Output;

END Umbrella.
```

6.3. Import statements

An **IMPORT** statement imports symbols from an interface for use in the importing module. Any type of module can have an **IMPORT** statement. It is a static error for an interface to import itself or for two or more interfaces to form an import cycle. There are two possible forms of the **IMPORT** statement; a given module can include either or both.

An **IMPORT** statement of the form:

IMPORT D_1, ..., D_n

declares the names D_i to be imported interfaces. The importing module can use any symbols from D_i, but must refer to each symbol S as a qualified identifier of the form D_i.S.

An **IMPORT** statement of the form:

FROM D **IMPORT** S_1, ..., S_n

introduces the symbols S_1, ..., S_n from the interface D into the scope of the importer. It does not introduce the name D; the importer must refer to S_i and not D.S_i. The importer cannot use any symbols from D that are not explicitly listed. It is legal to import the same interface in both forms of the **IMPORT** statement, in which case the listed symbols are available in both qualified and unqualified form.

Importing the name of an enumeration E via **FROM** D **IMPORT** E imports E's constants as well. Importing an interface provides access only to the symbols it defines, not the ones it imports via **FROM**.

All the type identifications visible in the scope of an imported interface are made visible to the importing scope (see 4.2, page 26).

6.4. Initialization

To *initialize* a module is to execute the variable initializations and statements in its main body. An implementation module is initialized when its initialization code has completed execution. An interface is initialized when the module that implements it is initialized.

When a program is started, its modules begin executing in an order constrained by the following:

> If an implementation module M imports an interface D and neither D nor its implementation indirectly imports any interface implemented by M, then D will be initialized before M. Otherwise, the initialization order is implementation-dependent.
>
> The main program module will always be initialized last.

A module `M` *imports* an interface `D` if `M` or any of the interfaces that `M` implements mention `D` in an `IMPORT` statement. A module `M` *indirectly imports* an interface `D` if `M` imports `D` or any interface that `M` imports is implemented by a module that indirectly imports `D`.

If a module is not indirectly imported by the main module, its initialization code is not executed.

6.5. Safety

By default, modules are considered unsafe unless the keyword `SAFE` precedes the declaration of the module.

An interface and its implementation are not required to have the same safety declaration. If an interface is safe and its implementation is unsafe, then the programmer, rather than the Modula-2+ implementation, is guaranteeing the safety of the implementation. Clients of such an interface may still assume that the implementation module is safe, and hence that it is safe to import procedures and variables from the interface.

The following are prohibited in safe modules:

Declaring an unsafe type (2.5, page 13).

Importing procedures, variables, or unsafe types from an unsafe interface.

Loopholing a value to a type that is not representation complete (2.5, page 13).

Passing a value whose syntactic type is not representation complete to a **VAR** or **VAR OUT BYTE**, **WORD**, **ARRAY OF BYTE**, or **ARRAY OF WORD** formal procedure parameter.

Changing the tag of a variant record or assigning variant records if any variant field of the record type is not representation complete (2.5, page 13).

DISPOSE applied to pointer types.

Narrowing (explicitly with **NARROW** or implicitly in an assignment) an **ADDRESS** value to a **POINTER** value.

COPY and **ZERO**.

For the purposes of safety, the body of an inline procedure appearing in a definition module is considered part of the corresponding implementation module.

7. Lexical structure

A lexical *token* is a keyword, identifier, literal, operator, or delimiter. Blanks, tabs, formfeeds, newlines, and comments separate tokens; no separation is required between operators or delimiters and other tokens. Formfeeds and newlines cannot appear within tokens. Blanks and tabs can occur inside quoted text and character literals. Inside quoted literals, blanks and tabs stand for themselves, but newlines do not. (For the lexical structure of literals, see 3.1, page 15.)

7.1. Keywords

```
AND             EXCEPT          LOOP            REPEAT
ARRAY           EXCEPTION       MOD             RETURN
BEGIN           EXIT            MODULE          SAFE
BITS            FINALLY         NOT             SET
BY              FOR             OF              THEN
CASE            FROM            OR              TO
CONST           IF              OUT             TRY
DEFINITION      IMPLEMENTATION  PASSING         TYPE
DIV             IMPLEMENTS      POINTER         TYPECASE
DO              IMPORT          PROCEDURE       UNTIL
ELSE            IN              RAISES          VAR
ELSIF           INLINE          RECORD          WHILE
END             LOCK            REF             WITH
```

7.2. Identifiers

An *identifier* is a sequence of letters and digits that is not a keyword. The first character of an identifier must be a letter. The following identifiers are predefined:

```
ABS             BITXOR    HALT        NUMBER
ADDR            BOOLEAN   HIGH        ORD
ADDRESS         BYTE      INC         PRINTF
ASSERT          BYTESIZE  INCL        RAISE
BITAND          CARDINAL  INTEGER     REAL
BITEXTRACT      CEILING   LAST        REFANY
BITINSERT       CHAR      LONGFLOAT   ROUND
BITNOT          COPY      LONGREAL    SCANF
BITOR           DEC       LOOPHOLE    TRUE
BITROTATELEFT   DISPOSE   LOW         TRUNC
BITROTATERIGHT  EXCL      MAX         TYPECODE
BITSET          FALSE     MIN         UNSIGNED
BITSHIFTLEFT    FIRST     NARROW      VAL
BITSHIFTRIGHT   FLOAT     NEW         WORD
BITSIZE         FLOOR     NIL         ZERO
```

7.3. Operators

Here are the characters or character pairs that serve as operators and delimiters:

```
+       <       #       =       ;       :=
-       >       {       }       ^       ,
*       <=      (       )       .       |
/       >=      [       ]       ..
```

7.4. Comments

Comments are arbitrary character sequences opened by (* and closed by *).
Comments can be nested and can extend over more than one line.

Appendix I. Syntax

Square brackets [ ] mean that the enclosed form is optional. Curly brackets { } mean that the enclosed form is repeated, possibly 0 times. Vertical bar | separates a list of choices. Terminals are either strings enclosed in quotes or reserved words written in all capital letters; non-terminals contain at least one lower-case letter.

```
terminals

    "+" "-"   "*" "/"  ":=" "&" "." "," ";" "(" "[" "{" "^" "=" "#" "<"
    ">" "<>" "<=" ">=" ".." ":" ")" "]" "}" "|"

    ident cardinal real char text

    AND         ELSIF          IMPORT PASSING   TRY
    ARRAY       END            IN     PROCEDURE TYPE
    BEGIN       EXCEPT         INLINE RAISES    TYPECASE
    BITS        EXCEPTION      LOCK   RECORD    UNTIL
    BY          EXIT           LOOP   REF       VAR
    CASE        FINALLY        MOD    REPEAT    WHILE
    CONST       FOR            MODULE RETURN    WITH
    DEFINITION  FROM           NOT    SAFE
    DIV         IF             OF     SET
    DO          IMPLEMENTATION OR     THEN
    ELSE        IMPLEMENTS     OUT    TO

productions

CompilationUnit =
    [SAFE] (
        MODULE ident [IMPLEMENTS Idents] ";"
            Imports Declarations [BEGIN Statements] END ident "." |
        IMPLEMENTATION MODULE ident ";"
            Imports Declarations [BEGIN Statements] END ident "." |
        DEFINITION MODULE ident ";"
            Imports Definitions END ident ".").

Imports = {Import}.

Definitions = {Definition}.

Import = [FROM ident] IMPORT Idents ";".

Block = Declarations [BEGIN Statements] END ident.

Declarations = {Declaration}

Definition =
    ConstDeclaration | TypeDeclaration | VarDeclaration |
    ExceptionDeclaration | ProcedureDefinition | INLINE ProcedureDeclaration

ProcedureDefinition = ProcedureHeading ["=" QualIdent] ";".

Declaration =
    ConstDeclaration | TypeDeclaration | VarDeclaration |
    ExceptionDeclaration | [INLINE] ProcedureDeclaration.

ConstDeclaration = CONST {ident "=" Expression ";"}.

TypeDeclaration = TYPE {QualIdent ["=" Type] ";"}.

VarDeclaration = VAR {Idents ":" Type ["=" QualIdent] ";"}.

ExceptionDeclaration =
    EXCEPTION {Idents ["(" QualIdent ")"] ["=" QualIdent] ";"}.

ProcedureDeclaration = [INLINE] ProcedureHeading ";" Block ";".

ProcedureHeading =
    PROCEDURE ident "(" Formals ")"
        [":" QualIdent] [RAISES "{" QualIdentsOrEmpty "}"].

Formals = [Formal {";" Formal}].

Formal =
```

```
    | % empty
    Mode Idents ":" [ARRAY OF] QualIdent [":=" Expression].

Mode = VAR [IN | OUT].

Statements = [Statement { ";" Statement }].

Statement =
    EmptyStatement | AssignmentOrProcCall | CaseStatement |
    ExitStatement | ForStatement | IfStatement | LockStatement |
    LoopStatement | RepeatStatement | ReturnStatement | TryStatement |
    TypeCaseStatement | WhileStatement | WithStatement.

AssignmentOrProcCall = Expression [":=" Expression].

CaseStatement = CASE Expression OF Cases [ELSE Statements] END.

Cases = [Case {"|" Case}].

Case =
    | %empty
    CaseLabels ":" Statements.

CaseLabels = CaseLabel {"," CaseLabel}.

CaseLabel = Expression [".." Expression].

ExitStatement = EXIT.

ForStatement =
    FOR ident ":=" Expression TO Expression [BY Expression]
        DO Statements END.

IfStatement =
    IF Expression THEN Statements
        {ELSIF Expression THEN Statements}
        [ELSE Statements]  END.

LockStatement = LOCK Expression DO Statements END.

LoopStatement = LOOP Statements END.

RepeatStatement = REPEAT Statements UNTIL Expression.

ReturnStatement = RETURN [Expression].

TryStatement = TRY Statements (Except | Finally | Passing) END.
Except       = EXCEPT Handlers [ELSE Statements].
Passing      = PASSING "{" QualIdentsOrEmpty "}"
Finally      = FINALLY Statements.

Handlers = [Handler {"|" Handler}].

Handler =
    | %empty
    QualIdents ["(" QualIdent ")"] ":" Statements.

TypeCaseStatement = TYPECASE Expression OF Handlers [ELSE Statements] END.

WhileStatement = WHILE Expression DO Statements END.

WithStatement = WITH Expression DO Statements END.

Expression = E1.

E1   = E2 {OR E2}.

E2   = E3 {AND E3}.
```

```
E3   = {NOT} E4.

E4   = E5 {E4Op E5}.
E4Op = "=" | "#" | "<"  | "<=" | ">" | ">=" | IN.

E5   = E6 {E5Op E6}.
E5Op = "+" | "-".

E6   = E7 {E6Op E7}.
E6Op = "*" | "/" | DIV | MOD.

E7   = {E7Op} E8.
E7Op = "-" | "+".

E8 = E9 {"^"                                |
         "." ident                          |
         "(" Expressions ")"                |
         "[" Expressions "]"                |
         "{" SetElements "}"}.

E9 = integer | unsigned | real | longreal | char | text | ident |
     "(" Expression ")".

SetElements = [SetElement {"," SetElement}].

SetElement = Expression [".." Expression].

Expressions = [Expression {"," Expression}].

Type =
    ArrayType | BitsType | EnumerationType | PointerType |
    ProcedureType | QualIdent | RecordType | RefType | SetType |
    SubrangeType.

ArrayType = ARRAY ArrayIndexTypes OF Type.

ArrayIndexTypes = [Type {"," Type}].

BitsType = BITS Expression FOR Type.

EnumerationType = "(" IdentsOrEmpty ")".

PointerType = POINTER TO Type.

ProcedureType =
    PROCEDURE ["(" TFormals ")"] [":" QualIdent]
        [RAISES "{" QualIdentsOrEmpty "}"].

EmptyTFormals = .

TFormals = [TFormal {"," TFormal}].

TFormal = Mode [ARRAY OF] QualIdent [":=" Expression].

RecordType = RECORD RecordFields END.

RecordFields = [RecordField {";" RecordField}].

RecordField =
    | % empty
    Idents ":" Type |
    CASE Ident [(":" QualIdent | "." Ident |)] OF Variants
        [ELSE RecordFields] END.

Variants = [Variant {"|" Variant}].

Variant =
    | % empty
```

```
    CaseLabels ":" RecordFields.

RefType = REF [Type].

SetType = SET OF Type.

SubrangeType = "[" Expression ".." Expression "]".

QualIdent = ident ["." ident].

QualIdentsOrEmpty = [QualIdents].

QualIdents = QualIdent {"," QualIdent}.

IdentsOrEmpty = [Idents].

Idents = ident {"," ident}.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%% Tokens %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

terminals

   " "     "#"     "&"     "'"     "("     ")"     "{"     "}"     "["     "]"
   "*"     "+"     ","     "-"     "."     "/"     ":"     ";"     "<"     "="
   ">"     "\"     "^"     "_"     "|"     "\t"    "\n"    "\f"    "\r"    "\b"
   DQUOTE

    "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"

    "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"

productions

Token =
    Identifier
  | Operator
  | Punctuation
  | CharacterLiteral
  | TextLiteral
  | Cardinal
  | Real
  | Comment.

Identifier = Letter {Letter | Digit}.

Operator =
    "+"      | "-"      | "*"      | "/"      | "."      | "^"      |
    ":" "="  | "="      | "#"      | "<"      | "<" "="  | ">" "=" |
    ">"

Punctuation =
    "," | ";" | "|" | ":" | "." "."    |
    "(" | ")" | "{" | "}" | "[" | "]".

CharacterLiteral = "'"  (Character | Escape) "'".

TextLiteral      = DQUOTE  {Character | Escape} DQUOTE.

Escape =
    "\" "n"   | "\" "t"     | "\" "r"     | "\" "f"    |
    "\" "b"   | "\" "\"     | "\" "'"     | "\" DQUOTE |
    "\" octalDigit octalDigit octalDigit.

Comment = "(" "*" {Comment | Character | space} "*" ")".
```

```
Cardinal  =
    Digit      {Digit}          |
    HexDigit   {HexDigit}   "H" |
    OctalDigit {OctalDigit} "B".

Real      = Digit {Digit} "." {Digit} [Exponent].
Exponent  =  ("E" | "e" | "D" | "d") [["+" | "-"] Digit {Digit}].

Character = Letter | Digit | Special | " ".

HexDigit =
    Digit |
    "A" | "B" | "C" | "D" | "E" | "F" |
    "a" | "b" | "c" | "d" | "e" | "f".

Digit = OctalDigit | "8" | "9".

OctalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" |"7".

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".

Space = " " | "\t" | "\n" | "\r" | "\f" | "\b".

Special =
    "#"  | "&"  | "("  | ")"  | "*"  | "+"  | ","  | "-"  | "."  | "/" |
    ":"  | ";"  | "<"  | "="  | ">"  | "["  | "\"  | "]"  | "^"  | "|" |
    "{"  | "}"  | "'"  | DQUOTE |
  % all remaining ASCII characters
  .
```

References

[1] Paul Rovner, Roy Levin, John Wick.
On Extending Modula-2 For Building Large, Integrated Systems.
Technical Report Research Report 3, Digital Systems Research Center, January, 1985.

[2] N. Wirth.
Programming in Modula-2.
Springer-Verlag, Third Edition, 1985.

Index