# Modula–3 Language Definition

Luca Cardelli, James Donahue*, Lucille Glassman,
Mick Jordan, Bill Kalsow, Greg Nelson

DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

*The language designer should be familiar with many alternative features designed by others, and should have excellent judgment in choosing the best and rejecting any that are mutually inconsistent... One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation.*
*—C.A.R. Hoare*

Modula–3 is a systems programming language developed by Digital Equipment Corporation and Olivetti between 1986 and 1990. It emphasizes safety and simplicity, and supports interfaces, objects, generics, lightweight threads of control, the isolation of unsafe code, garbage collection, exceptions, and subtyping. Modula–3 is based on—but is not compatible with—the Modula–2 language designed by Niklaus Wirth, and the Modula–2+ language developed at Digital. Two books on the language have been published: *Systems Programming with Modula–3* by Greg Nelson (Prentice-Hall, 1991), and *Modula–3* by Sam Harbison (Prentice-Hall, 1992). On-line discussions of the language may be found in the Usenet news group `comp.lang.modula3`.

A Modula–3 compiler and a collection of tools and libraries is distributed free by the DEC Systems Research Center in Palo Alto. For more information, send E-mail to `m3-request@src.dec.com`.

## Contents

A previous version of this paper appears in *Systems Programming with Modula-3*, Greg Nelson, ed., © Prentice-Hall, 1991, and under copyright of Digital Equipment Corporation and Ing. C. Olivetti and C., SpA. This version appears with permission. James Donahue's current affiliation is Documentum, Inc., 5724 West Las Positas, Pleasanton, CA 94588. This paper was edited and formatted by Sam Harbison, Pine Creek Software.

## 1 Definitions

A Modula–3 program specifies a computation that acts on a sequence of digital components called *locations*. A *variable* is a set of locations that represents a mathematical value according to a convention determined by the variable's *type*. If a value can be represented by some variable of type `T`, then we say that the value is a *member* of `T` and `T` *contains* the value.

An *identifier* is a symbol declared as a name for a variable, type, procedure, etc. The region of the program over which a declaration applies is called the *scope* of the declaration. Scopes can be nested. The meaning of an identifier is determined by the smallest enclosing scope in which the identifier is declared.

An *expression* specifies a computation that produces a value or variable. Expressions that produce variables are called *designators*. A designator can denote either a variable or the value of that variable, depending on the context. Some designators are *readonly*, which means that they cannot be used in contexts that might change the value of the variable. A designator that is not readonly is called *writable*. Expressions whose values can be determined statically are called *constant expressions*; they are never designators.

A *static error* is an error that the implementation must detect before program execution. Violations of the language definition are static errors unless they are explicitly classified as runtime errors.

A *checked runtime error* is an error that the implementation must detect and report at runtime. The method for reporting such errors is implementation-dependent. (If the implementation maps them into exceptions, then a program could handle these exceptions and continue.)

An *unchecked runtime error* is an error that is not guaranteed to be detected, and can cause the subsequent behavior of the computation to be arbitrary. Unchecked runtime errors can occur only in unsafe modules.

## 2 Types

*I am the voice of today, the herald of tomorrow...*
*I am the leaden army that conquers the world—I am TYPE.*
*—Frederic William Goudy*

Modula–3 uses structural equivalence, instead of the name equivalence of Modula-2. Two types are the same if their definitions become the same when expanded; that is, when all constant expressions are replaced by their values and all type names are replaced by their definitions. In the case of recursive types, the expansion is the infinite limit of the partial expansions. A type expression is generally allowed wherever a type is required.

A type is *empty* if it contains no values. For example, `[1..0]` is an empty type. Empty types can be used to build non-empty types (for example, `SET OF [1..0]`, which is not empty because it contains the empty set). It is a static error to declare a variable of an empty type.

Every expression has a statically-determined type, which contains every value the expression can produce. The type of a designator is the type of the variable it produces.

Assignability and type compatibility are defined in terms of a single syntactically specified subtype relation with the property that if `T` is a subtype of `U`, then every member of `T` is a member of `U`. The subtype relation is reflexive and transitive.

Every expression has a unique type, but a value can be a member of many types. For example, the value 6 is a member of both `[0..9]` and `INTEGER`. It would be ambiguous to talk about "the type of a value". Thus the phrase "type of `x`" means "type of the expression `x`", while "`x` is a member of `T`" means "the value of `x` is a member of `T`".

However, there is one sense in which a value can be said to have a type: every object or traced reference value includes a code for a type, called the *allocated type* of the reference value. The allocated type is tested by `TYPECASE` (Section 3.18).

## 2.1 Ordinal types

There are three kinds of ordinal types: enumerations, subranges, and `INTEGER`. An enumeration type is declared like this:

```
TYPE T = {id_1, ..., id_n}
```

where the `id`'s are distinct identifiers. The type `T` is an ordered set of n values; the expression `T.id`$_i$ denotes the $i$'th value of the type in increasing order. The empty enumeration `{ }` is allowed.

Integers and enumeration elements are collectively called *ordinal values*. The *base type* of an ordinal value `v` is `INTEGER` if `v` is an integer, otherwise it is the unique enumeration type that contains `v`.

A subrange type is declared like this:

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

where `Lo` and `Hi` are two ordinal values with the same base type, called the base type of the subrange. The values of `T` are all the values from `Lo` to `Hi` inclusive. `Lo` and `Hi` must be constant expressions (Section 6.15). If `Lo` exceeds `Hi`, the subrange is empty.

The operators `ORD` and `VAL` convert between enumerations and integers. The operators `FIRST`, `LAST`, and `NUMBER` applied to an ordinal type return the first element, last element, and number of elements, respectively (Section 6.13).

Here are the predeclared ordinal types:

| | |
|---|---|
| `INTEGER` | All integers represented by the implementation |
| `CARDINAL` | Subrange `[0..LAST(INTEGER)]` |
| `BOOLEAN` | The enumeration `{FALSE, TRUE}` |
| `CHAR` | An enumeration containing at least 256 elements |

The first 256 elements of type `CHAR` represent characters in the ISO-Latin-1 code, which is an extension of ASCII. The language does not specify the names of the elements of the `CHAR` enumeration. The syntax for character literals is in Section 6.5. `FALSE` and `TRUE` are predeclared synonyms for `BOOLEAN.FALSE` and `BOOLEAN.TRUE`.

Each distinct enumeration type introduces a new collection of values, but a subrange type reuses the values from the underlying type. For example:

```
TYPE
  T1 = {A, B, C};
  T2 = {A, B, C};
  U1 = [T1.A..T1.C];
  U2 = [T1.A..T2.C];  (* sic *)
  V =  {A, B}
```

`T1` and `T2` are the same type, since they have the same expanded definition. In particular, `T1.C = T2.C` and therefore `U1` and `U2` are also the same type. But the types `T1` and `U1` are distinct, although they contain the same values, because the expanded definition of `T1` is an enumeration while the expanded definition of `U1` is a subrange. The type `V` is a third type whose values `V.A` and `V.B` are not related to the values `T1.A` and `T1.B`.

## 2.2 Floating-point types

There are three floating point types, which in order of increasing range and precision are `REAL`, `LONGREAL`, and `EXTENDED`. The properties of these types are specified by required interfaces in Section 9.4.

## 2.3 Arrays

An *array* is an indexed collection of component variables, called the *elements* of the array. The indexes are the values of an ordinal type, called the *index type* of the array. The elements all have the same size and the same type, called the *element type* of the array.

There are two kinds of array types, *fixed* and *open*. The length of a fixed array is determined at compile time. The length of an open array type is determined at runtime, when it is allocated or bound. The length cannot be changed thereafter.

The *shape* of a multidimensional array is the sequence of its lengths in each dimension. More precisely, the shape of an array is its length followed by the shape of any of its elements; the shape of a non-array is the empty sequence.

Arrays are assignable if they have the same element type and shape. If either the source or target of the assignment is an open array, a runtime shape check is required.

A fixed array type declaration has the form:

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

where `Index` is an ordinal type and `Element` is any type other than an open array type. 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]` designates the element of `a` whose position corresponds to the position of `i` in `Index`. For example, consider the declarations:

```
VAR a:= ARRAY [1..3] OF
  REAL {1.0, 2.0, 3.0};
VAR b: ARRAY [-1..1] OF REAL := a;
```

Now `a = b` is `TRUE`; yet `a[1] = 1.0` while `b[1] = 3.0`. The interpretation of indexes is determined by an array's type, not its value; the assignment `b := a` changes `b`'s value, not its type. (This example uses variable initialization, Section 4.3, and array constructors, Section 6.8.)

An expression of the form:

ARRAY $\texttt{Index}_1$, ......, $\texttt{Index}_n$ OF Element

is shorthand for:

ARRAY $\texttt{Index}_1$ OF ... OF ARRAY $\texttt{Index}_n$ OF Element

This shorthand is eliminated from the expanded type definition used to define structural equivalence. An expression of the form `a[`$i_1$, ..., $i_n$`]` is shorthand for `a[`$i_1$`]`...`[`$i_n$`]`.

An open array type declaration has the form:

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

where `Element` is any type. The values of `T` are arrays whose element type is `Element` and whose length is arbitrary. The index type of an open array is the integer subrange `[0..n-1]`, where `n` is the length of the array.

An open array type can be used only as the type of a formal parameter, the referent of a reference type, the element type of another open array type, or as the type in an array constructor.

Examples of array types:

```
TYPE
  Transform = ARRAY [1..3], [1..3]
    OF REAL;
  Vector = ARRAY OF REAL;
  SkipTable = ARRAY CHAR OF INTEGER
```

## 2.4 Records

A *record* is a sequence of named variables, called the *fields* of the record. Different fields can have different types. The name and type of each field is statically determined by the record's type. The expression `r.f` designates the field named `f` in the record `r`.

A record type declaration has the form:

```
TYPE T = RECORD FieldList END
```

where `FieldList` is a list of field declarations, each of which has the form:

```
fieldName: Type := default
```

where `fieldName` is an identifier, `Type` is any non-empty type other than an open array type, and `default` is a constant expression. The field names must be distinct. A record is a member of `T` if it has fields with the given names and types, in the given order, and no other fields. Empty records are allowed.

The constant `default` is a default value used when a record is constructed (Section 6.8) or allocated (Section 6.9). Either "`:= default`" or "`: Type`" can be omitted, but not both. If `Type` is omitted, it is taken to be the type of `default`. If both are present, the value of `default` must be a member of `Type`.

When a series of fields shares the same type and default, any `fieldName` can be a list of identifiers separated by commas. Such a list is shorthand for a list in which the type and default are repeated for each identifier. That is:

$f_1$, ..., $f_m$: Type := default

is shorthand for:

$f_1$: Type := default; ...;
$f_m$: Type := default

This shorthand is eliminated from the expanded definition of the type. The default values are included.

Examples of record types:

```
TYPE
  Time = RECORD
    seconds: INTEGER;
    milliseconds: [0..999]
  END;
  Alignment = {Left, Center, Right};
  TextWindowStyle = RECORD
    align := Alignment.Center;
    font := Font.Default;
    foreground := Color.Black;
    background := Color.White;
    margin, border := 2
  END
```

## 2.5 Packed types

A declaration of a packed type has the form:

```
TYPE T = BITS n FOR Base
```

where `Base` is a type and `n` is an integer-valued constant expression. The values of type `T` are the same as the values of type `Base`, but variables of type `T` that occur in records, objects, or arrays will occupy exactly n bits and be packed adjacent to the preceding field or element. For example, a variable of type

```
    ARRAY [0..255] OF BITS 1 FOR BOOLEAN
```

is an array of 256 Booleans, each of which occupies one bit of storage.

If the first field of an object subtype has a BITS FOR type, then it will not necessarily be packed adjacent to the last field of the supertype.

The values allowed for n are implementation-dependent. An illegal value for n is a static error. The legality of a packed type can depend on its context; for example, an implementation could prohibit packed integers from spanning word boundaries.

## 2.6 Sets

A *set* is a collection of values taken from some ordinal type. A set type declaration has the form:

```
    TYPE T = SET OF Base
```

where Base is an ordinal type. The values of T are all sets whose elements have type Base. For example, a variable whose type is SET OF [0..1] can assume the following values:

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

Implementations are expected to use the same representation for a SET OF T as for an ARRAY T OF BITS 1 FOR BOOLEAN. Hence, programmers should expect SET OF [0..1023] to be practical, but not SET OF INTEGER.

## 2.7 References

A *reference* value is either NIL or the address of a variable, called the referent.

A reference type is either *traced* or *untraced.* When all traced references to a piece of allocated storage are gone, the implementation reclaims the storage. Two reference types are of the same *reference class* if they are both traced or both untraced. A general type is traced if it is a traced reference type, a record type any of whose field types is traced, an array type whose element type is traced, or a packed type whose underlying unpacked type is traced.

A declaration for a traced reference type has the form:

```
    TYPE T = REF Type
```

where Type is any type. The values of T are traced references to variables of type Type, which is called the *referent type* of T.

A declaration for an untraced reference type has the form:

```
    TYPE T = UNTRACED REF Type
```

where Type is any untraced[1] type. The values of T are the untraced references to variables of type Type.

In both the traced and untraced cases, the keyword REF can optionally be preceded by "BRANDED b" where b is a text constant called the *brand.* Brands distinguish types that would otherwise be the same; they have no other semantic effect. All brands in a program must be distinct. If BRANDED is present and b is absent, the implementation automatically supplies a unique value for b. Explicit brands are useful for persistent data storage.

The following reference types are predeclared:

| | |
|---|---|
| REFANY | Contains all traced references |
| ADDRESS | Contains all untraced references |
| NULL | Contains only NIL |

The TYPECASE statement (Section 3.18) can be used to test the referent type of a REFANY or object, but there is no such test for an ADDRESS.

Examples of reference types:

```
TYPE
  TextLine = REF ARRAY OF CHAR;
  ControllerHandle =
    UNTRACED REF RECORD
    status: BITS 8 FOR [0..255];
    filler: BITS 12 FOR [0..0];
    pc: BITS 12 FOR [0..4095]
  END;
  T = BRANDED "ANSI" REF INTEGER;
  Apple = BRANDED REF INTEGER;
  Orange = BRANDED REF INTEGER;
```

## 2.8 Procedures

A *procedure* is either NIL or a triple consisting of:

- the *body*, which is a statement,
- the *signature*, which specifies the procedure's formal arguments, result type, and raises set (the set of exceptions that the procedure can raise),
- the *environment*, which is the scope with respect to which variable names in the body will be interpreted.

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 *top-level* procedure is a procedure declared in the outermost scope of a module. Any other procedure is a *local* procedure. A local procedure can be passed as a parameter but not assigned, since in a stack implementation a local procedure becomes invalid when the frame for the procedure containing it is popped.

A *procedure constant* is an identifier declared as a procedure. (As opposed to a procedure variable, which is a variable declared with a procedure type.)

A procedure type declaration has the form:

```
    TYPE T = PROCEDURE sig
```

where sig is a signature specification, which has the form:

[1] This restriction is lifted in unsafe modules.

```
(formal$_1$; ...; formal$_n$): R RAISES S
```

where

- Each `formal`$_i$ is a formal parameter declaration, as described below.
- `R` is the result type, which can be any type but an open array type. The "`: R`" can be omitted, making the signature that of a proper procedure.
- `S` is the raises set, which is either an explicit set of exceptions with the syntax `{E`$_1$`, ..., E`$_m$`}`, or the symbol `ANY` representing the set of all exceptions. If "`RAISES S`" is omitted, "`RAISES {}`" is assumed.

A formal parameter declaration has the form

```
Mode Name: Type := Default
```

where

- `Mode` is a parameter mode, which can be `VALUE`, `VAR`, or `READONLY`. If `Mode` is omitted, it defaults to `VALUE`.
- `Name` is an identifier that names the parameter. The parameter names must be distinct.
- `Type` is the type of the parameter.
- `Default` is a constant expression, the default value for the parameter. If `Mode` is `VAR`, "`:= Default`" must be omitted, otherwise either "`:= Default`" or "`: Type`" can be omitted, but not both. If `Type` is omitted, it is taken to be the type of `Default`. If both are present, the value of `Default` must be a member of `Type`.

When a series of parameters share the same mode, type, and default, `name`$_i$ can be a list of identifiers separated by commas. Such a list is shorthand for a list in which the mode, type, and default are repeated for each identifier. That is:

```
Mode v$_1$, ..., v$_n$: Type := Default
```

is shorthand for:

```
Mode v$_1$: Type := Default; ...; Mode v$_n$:
Type := Default
```

This shorthand is eliminated from the expanded definition of the type. The default values are included.

A procedure value `P` is a member of the type `T` if it is `NIL` or its signature is *covered* by the signature of `T`, where `signature`$_1$ covers `signature`$_2$ if:

- They have the same number of parameters, and corresponding parameters have the same type and mode.
- They have the same result type, or neither has a result type.
- The raises set of `signature`$_1$ contains the raises set of `signature`$_2$.

The parameter names and defaults affect the type of a procedure, but not its value. For example, consider the declarations:

```
PROCEDURE P(txt: TEXT := "P") =
  BEGIN
    Wr.PutText(Stdio.stdout, txt)
  END P;
VAR q:PROCEDURE(txt:TEXT:="Q") := P;
```

Now `P = q` is `TRUE`, yet `P()` prints "`P`" and `q()` prints "`Q`". The interpretation of defaulted parameters is determined by a procedure's type, not its value; the assignment `q := P` changes `q`'s value, not its type.

Examples of procedure types:

```
TYPE
  Integrand = PROCEDURE (x: REAL):
REAL;
  Integrator = PROCEDURE(
    f:Integrand; lo,hi:REAL): REAL;
  TokenIterator = PROCEDURE(
    VAR t: Token) RAISES {TokenErr};
  RenderProc = PROCEDURE(
    scene: REFANY;
    READONLY t:Transform:=Identity);
```

In a procedure type, `RAISES` binds to the closest preceding `PROCEDURE`. That is, these parentheses are required:

```
TYPE T = PROCEDURE ()
  : (PROCEDURE ()) RAISES {}
```

## 2.9 Objects

An *object* is either `NIL` or a reference to a data record paired with a method suite, which is a record of procedures that will accept the object as a first argument.

An object type determines the types of a prefix of the fields of the data record, as if "`OBJECT`" were "`REF RECORD`". But in the case of an object type, the data record can contain additional fields introduced by subtypes of the object type. Similarly, the object type determines a prefix of the method suite, but the suite can contain additional methods introduced by subtypes.

If `o` is an object, then `o.f` designates the data field named `f` in `o`'s data record. If `m` is one of `o`'s methods, an invocation of the form `o.m(...)` denotes an execution of `o`'s `m` method (Section 3.2). An object's methods can be invoked, but not read or written.

If `T` is an object type and `m` is the name of one of `T`'s methods, then `T.m` denotes `T`'s `m` method. This notation makes it convenient for a subtype method to invoke the corresponding method of one of its supertypes.

A field or method in a subtype masks any field or method with the same name in the supertype. To access such a masked field, use `NARROW` to view the subtype variable as a member of the supertype, as illustrated later in this section.

Object assignment is reference assignment. Objects cannot be dereferenced, since the static type of an object variable does not determine the type of its data record. To copy the data record of one object into another, the fields must be assigned individually.

There are two predeclared object types:

| | |
|---|---|
| `ROOT` | The traced object type with no fields or methods |
| `UNTRACED ROOT` | The untraced object type with no fields or methods |

The declaration of an object type has the form:

```
TYPE T = ST OBJECT Fields
  METHODS Methods
  OVERRIDES Overrides
END
```

where `ST` is an optional supertype, `Fields` is a list of field declarations, exactly as in a record type, `Methods` is a list of *method declarations* and `Overrides` is a list of *method overrides*. The fields of `T` consist of the fields of `ST` followed by the fields declared in `Fields`. The methods of `T` consist of the methods of `ST` modified by `Overrides` and followed by the methods declared in `Methods`. `T` has the same reference class as `ST`.

The names introduced in `Fields` and `Methods` must be distinct from one another and from the names overridden in `Overrides`. If `ST` is omitted, it defaults to `ROOT`. If `ST` is untraced, then the fields must not include traced types.[1] If `ST` is declared as an opaque type (Section 4.6), the declaration of `T` is legal only in scopes where `ST`'s concrete type is known to be an object type.

The keyword `OBJECT` can optionally be preceded by "`BRANDED`" or by "`BRANDED b`", where `b` is a text literal. The meaning is the same as in non-object reference types.

A method declaration has the form:

```
m sig := proc
```

where `m` is an identifier, `sig` is a procedure signature, and `proc` is a top-level procedure constant. It specifies that `T`'s `m` method has signature `sig` and value `proc`. If "`:= proc`" is omitted, "`:= NIL`" is assumed. If `proc` is non-nil, its first parameter must have mode `VALUE` and type some supertype of `T`, and dropping its first parameter must result in a signature that is covered by `sig`.

A method override has the form:

```
m := proc
```

where `m` is the name of a method of the supertype `ST` and `proc` is a top-level procedure constant. It specifies that the `m` method for `T` is `proc`, rather than `ST.m`. If `proc` is non-nil, its first parameter must have mode `VALUE` and type some supertype of `T`, and dropping its first parameter must result in a signature that is covered by the signature of `ST`'s `m` method.

[1] This restriction is lifted in unsafe modules.

As examples, consider the following declarations:

```
TYPE
  A = OBJECT a: INTEGER; METHODS p()
END;
  AB = A OBJECT b: INTEGER END;

PROCEDURE Pa(self: A) = ... ;
PROCEDURE Pab(self: AB) = ... ;
```

The procedures `Pa` and `Pab` are candidate values for the `p` methods of objects of types `A` and `AB`. For example:

```
TYPE T1 = AB OBJECT
  OVERRIDES p := Pab END
```

declares a type with an `AB` data record and a `p` method that expects an `AB`. `T1` is a valid subtype of `AB`. Similarly,

```
TYPE T2 = A OBJECT
  OVERRIDES p := Pa END
```

declares a type with an `A` data record and a method that expects an `A`. `T2` is a valid subtype of `A`. A more interesting example is:

```
TYPE T3 = AB OBJECT
  OVERRIDES p := Pa END
```

which declares a type with an `AB` data record and a `p` method that expects an `A`. Since every `AB` is an `A`, the method is not too choosy for the objects in which it will be placed. `T3` is a valid subtype of `AB`. In contrast,

```
TYPE T4 = A OBJECT
  OVERRIDES p := Pab END
```

attempts to declare a type with an `A` data record and a method that expects an `AB`; since not every `A` is an `AB`, the method is too choosy for the objects in which it would be placed. The declaration of `T4` is a static error.

The following example illustrates the difference between declaring a new method and overriding an existing method. After the declarations

```
TYPE
  A = OBJECT METHODS m() := P END;
  B = A OBJECT OVERRIDES m := Q END;
  C = A OBJECT METHODS m() := Q END;
VAR
  a := NEW(A); b := NEW(B);
  c := NEW(C);
```

we have that

```
a.m() activates P(a)
b.m() activates Q(b)
c.m() activates Q(c)
```

So far there is no difference between overriding and extending. But `c`'s method suite has two methods, while `b`'s has only one, as can be revealed if `b` and `c` are viewed as members of type `A`:

```
NARROW(b, A).m() activates Q(b)
NARROW(c, A).m() activates P(c)
```

Here `NARROW` is used to view a variable of a subtype as a value of its supertype. It is more often used for the opposite purpose, when it requires a runtime check (see Section 6.13).

The last example uses object subtyping to define reusable queues. First the interface:

```
TYPE
  Queue = RECORD
    head, tail: QueueElem END;
  QueueElem = OBJECT link: QueueElem
END;
PROCEDURE Insert(
  VAR q: Queue; x: QueueElem);
PROCEDURE Delete(
  VAR q: Queue): QueueElem;
PROCEDURE Clear(
  VAR q: Queue);
```

Then an example client:

```
TYPE
  IntQueueElem = QueueElem OBJECT
    val: INTEGER END;
VAR
  q: Queue;
  x: IntQueueElem;
  ...
  Clear(q);
  x := NEW(IntQueueElem, val := 6);
  Insert(q, x);
  ...
  x := Delete(q)
```

Passing `x` to `Insert` is safe, since every `IntQueueElem` is a `QueueElem`. Assigning the result of `Delete` to `x` cannot be guaranteed valid at compile-time, since other subtypes of `QueueElem` can be inserted into `q`, but the assignment will produce a checked runtime error if the source value is not a member of the target type. Thus `IntQueueElem` bears the same relation to `QueueElem` as `[0..9]` bears to `INTEGER`.

## 2.10 Subtyping rules

We write `T <: U` to indicate that `T` is a subtype of `U` and `U` is a supertype of `T`.

If `T <: U`, then every value of type `T` is also a value of type `U`. The converse does not hold: for example, a record or array type with packed fields contains the same values as the corresponding type with unpacked fields, but there is no subtype relation between them. This section presents the rules that define the subtyping relation.

For ordinal types `T` and `U`, we have `T <: U` if they have the same basetype and every member of `T` is a member of `U`. That is, subtyping on ordinal types reflects the subset relation on the value sets.

For array types,

```
(ARRAY OF)^m ARRAY J_1 OF ... ARRAY J_n OF
   ARRAY K_1 OF ... ARRAY K_p OF T
<: (ARRAY OF)^m (ARRAY OF)^n
     ARRAY I_1 OF ... ARRAY I_p OF T
```

if `NUMBER(I_i) = NUMBER(K_i)` for $i = 1, \ldots, p$.

That is, an array type `A` is a subtype of an array type `A'` if they have the same ultimate element type, the same number of dimensions, and, for each dimension, either both are open (as in the first *m* dimensions above), or `A` is fixed and `A'` is open (as in the next *n* dimensions), or both are fixed and have the same size (as in the last *p* dimensions).

```
NULL <: REF T <: REFANY
NULL <: UNTRACED REF T <: ADDRESS
```

That is, `REFANY` and `ADDRESS` contain all traced and untraced references, respectively, and `NIL` is a member of every reference type. These rules also apply to branded types.

```
NULL <: PROCEDURE(A): R RAISES S
```
for any `A`, `R`, and `S`.

That is, `NIL` is a member of every procedure type.

```
PROCEDURE(A): Q RAISES E <:
  PROCEDURE(B): R RAISES F
```
if signature `(B): R RAISES F` covers signature `(A): Q RAISES E`

That is, for procedure types, `T <: T'` if they are the same except for parameter names, defaults, and the raises set, and the raises set for `T` is contained in the raises set for `T'`.

```
ROOT <: REFANY
UNTRACED ROOT <: ADDRESS
NULL <: T OBJECT ... END <: T
```

That is, every object is a reference, `NIL` is a member of every object type, and every subtype is included in its supertype. The third rule also applies to branded types.

`BITS n FOR T <: T` and
`T <: BITS n FOR T`

That is, `BITS FOR T` has the same values as `T`.

`T <: T` for all `T`
`T <: U` and `U <: V` implies `T <: V` for all `T, U, V`.

That is, `<:` is reflexive and transitive.

Note that `T <: U` and `U <: T` does not imply that `T` and `U` are the same, since the subtype relation is unaffected by parameter names, default values, and packing.

For example, consider:

```
TYPE
  T = [0..255];
  U = BITS 8 FOR [0..255];
  AT = ARRAY OF T;
  AU = ARRAY OF U;
```

The types T and U are subtypes of one another but are not the same. The types AT and AU are unrelated by the subtype relation.

## 2.11 Predeclared opaque types

The language predeclares the two types:

```
TEXT <: REFANY
MUTEX <: ROOT
```

which represent text strings and mutual exclusion semaphores, respectively. These are opaque types as defined in Section 4.6. Their properties are specified in the required interfaces `Text` (Section 9.1) and `Thread` (Section 9.2).

# 3 Statements

*Look into any carpenter's tool-bag and see how many different hammers, chisels, planes and screw-drivers he keeps there—not for ostentation or luxury, but for different sorts of jobs.*
*—Robert Graves and Alan Hodge*

Executing a statement produces a computation that can halt (normal outcome), raise an exception, cause a checked runtime error, or loop forever. If the outcome is an exception, it can optionally be paired with an argument.

We define the semantics of EXIT and RETURN with exceptions called the *exit-exception* and the *return-exception*. The exit-exception takes no argument; the return-exception takes an argument of arbitrary type. Programs cannot name these exceptions explicitly.

Implementations should speed up normal outcomes at the expense of exceptions (except for the return-exception and exit-exception). Expending a thousand instructions per exception raised to save one instruction per procedure call would be reasonable.

If an expression is evaluated as part of the execution of a statement, and the evaluation raises an exception, then the exception becomes the outcome of the statement.

The empty statement is a no-op. In this report, empty statements are written `(*skip*)`.

## 3.1 Assignment

To specify the typechecking of assignment statements we need to define "assignable", which is a relation between types and types, between expressions and variables, and between expressions and types.

A type T is *assignable* to a type U if:

- T <: U, or
- U <: T and T is an array or a reference type other than ADDRESS[1], or
- T and U are ordinal types with at least one member in common.

An expression e is *assignable* to a variable v if:

- the type of e is assignable to the type of v, and
- the value of e is a member of the type of v, is not a local procedure, and if it is an array, then it has the same shape as v.

The first point can be checked statically; the others generally require runtime checks. Since there is no way to determine statically whether the value of a procedure parameter is local or global, assigning a local procedure is a runtime rather than a static error.

An expression e is *assignable* to a type T if e is assignable to some variable of type T. (If T is not an open array type, this is the same as saying that e is assignable to any variable of type T.)

An assignment statement has the form:

```
v := e
```

where v is a writable designator and e is an expression assignable to the variable designated by v. The statement sets v to the value of e. The order of evaluation of v and e is undefined, but e will be evaluated before v is updated. In particular, if v and e are overlapping subarrays (Section 6.3), the assignment is performed in such a way that no element is used as a target before it is used as a source.

Examples of assignments:

```
VAR x: REFANY;
    a: REF INTEGER;
    b: REF BOOLEAN;

a := b;  (* static error *)
x := a;  (* no possible error *)
a := x;  (* possible checked
            runtime error *)
```

The same comments would apply if x had an ordinal type with non-overlapping subranges a and b, or if x had an object type and a and b had incompatible subtypes. The type ADDRESS is treated differently from other reference types, since a runtime check cannot be performed on the assignment of raw addresses. For example:

[1] This restriction is lifted in unsafe modules.

```
VAR x: ADDRESS;
    a: UNTRACED REF INTEGER;
    b: UNTRACED REF BOOLEAN;

a := b;  (* static error *)
x := a;  (* no possible error *)
a := x;  (* static error in
            safe modules *)
```

### 3.2 Procedure call

A procedure call has the form:

```
P(Bindings)
```

where P is a procedure-valued expression and Bindings is a list of *keyword* or *positional* bindings. A keyword binding has the form name := actual, where actual is an expression and name is an identifier. A positional binding has the form actual, where actual is an expression. When keyword and positional bindings are mixed in a call, the positional bindings must precede the keyword bindings. If the list of bindings is empty, the parentheses are still required.

The list of bindings is rewritten to fit the signature of P's type as follows: First, each positional binding actual is converted and added to the list of keyword bindings by supplying the name of the i'th formal parameter, where actual is the i'th binding in Bindings. Second, for each parameter that has a default and is not bound after the first step, the binding name := default is added to the list of bindings, where name is the name of the parameter and default is its default value. The rewritten list of bindings must bind only formal parameters and must bind each formal parameter exactly once. For example, suppose that the type of P is

```
PROCEDURE(ch: CHAR; n: INTEGER := 0)
```

Then the following calls are all equivalent:

```
P('a', 0)
P('a')
P(ch := 'a')
P(n := 0, ch := 'a')
P('a', n := 0)
```

The call P() is illegal, since it doesn't bind ch. The call P(n := 0, 'a') is illegal, since it has a keyword parameter before a positional parameter.

For a READONLY or VALUE parameter, the actual can be any expression assignable to the type of the formal (except that the prohibition against assigning local procedures is relaxed). For a VAR parameter, the actual must be a writable designator whose type is the same as that of the formal, or, in case of a VAR array parameter, assignable to that of the formal. Designators are defined in Section 6.3.

A VAR formal is bound to the variable designated by the corresponding actual; that is, it is aliased. A VALUE formal is bound to a variable with an unused location and initialized to the value of the corresponding actual. A READONLY formal is treated as a VAR formal if the actual is a designator and the type of the actual is the same as the type of the formal (or an array type that is assignable to the type of the formal); otherwise it is treated as a VALUE formal.

Implementations are allowed to forbid VAR or READONLY parameters of packed types.

To execute the call, the procedure P and its arguments are evaluated, the formal parameters are bound, and the body of the procedure is executed. The order of evaluation of P and its actual arguments is undefined. It is a checked runtime error to call an undefined or NIL procedure.

It is a checked runtime error for a procedure to raise an exception not included in its raises set[1] or for a function procedure to fail to return a result.

A procedure call is a statement only if the procedure is proper. To call a function procedure and discard its result, use EVAL.

A procedure call can also have the form:

```
o.m(Bindings)
```

where o is an object and m names one of o's methods. This is equivalent to:

```
(o's m method) (o, Bindings)
```

### 3.3 Eval

An EVAL statement has the form:

```
EVAL e
```

where e is an expression. The effect is to evaluate e and ignore the result. For example:

```
EVAL Thread.Fork(p)
```

### 3.4 Block statement

A block statement has the form:

```
Decls BEGIN S END
```

where Decls is a sequence of declarations and S is a statement. The block introduces the constants, types, variables, and procedures declared in Decls and then executes S. The scope of the declared names is the block. (See Section 4.)

### 3.5 Sequential composition

A statement of the form:

$S_1; S_2$

executes $S_1$, and then if the outcome is normal, executes $S_2$. If the outcome of $S_1$ is an exception, $S_2$ is ignored.[2]

[1] If an implementation maps this runtime error into an exception, the exception is implicitly included in all RAISES clauses.

## 3.6 Raise

A RAISE statement without an argument has the form:

```
RAISE e
```

where e is an exception that takes no argument. The outcome of the statement is the exception e. A RAISE statement with an argument has the form:

```
RAISE e(x)
```

where e is an exception that takes an argument and x is an expression assignable to e's argument type. The outcome is the exception e paired with the argument x.

## 3.7 Try Except

A TRY-EXCEPT statement has the form:

```
TRY
  Body
EXCEPT
  id_1 (v_1) => Handler_1
| ...
| id_n (v_n) => Handler_n
ELSE Handler_0
END
```

where Body and each Handler are statements, each id names an exception, and each $v_i$ is an identifier. The "ELSE $Handler_0$" and each " $(v_i)$ " are optional. It is a static error for an exception to be named 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 $id_i$, then $Handler_i$ is executed. If Body raises any other exception and "ELSE $Handler_0$" is present, then it is executed. In either case, the outcome of the TRY statement is the outcome of the selected handler. If Body raises an unlisted exception and "ELSE $Handler_0$" is absent, then the outcome of the TRY statement is the exception raised by Body.

Each $(v_i)$ declares a variable whose type is the argument type of the exception $id_i$ and whose scope is $Handler_i$. When an exception $id_i$ paired with an argument x is handled, $v_i$ is initialized to x before $Handler_i$ is executed. It is a static error to include $(v_i)$ if exception $id_i$ does not take an argument.

If $(v_i)$ is absent, then $id_i$ can be a list of exceptions separated by commas, as shorthand for a list in which the rest of the handler is repeated for each exception. That is:

```
id_1, ..., id_n => Handler
```

is shorthand for:

```
id_1 => Handler; ...; id_n => Handler
```

It is a checked runtime error to raise an exception outside the dynamic scope of a handler for that exception. A "TRY EXCEPT ELSE" counts as a handler for all exceptions.

## 3.8 Try Finally

A statement of the form:

```
TRY S_1 FINALLY S_2 END
```

executes statement $S_1$ and then statement $S_2$. If the outcome of $S_1$ is normal, the TRY statement is equivalent to $S_1$; $S_2$. If the outcome of $S_1$ is an exception and the outcome of $S_2$ is normal, the exception from $S_1$ is re-raised after $S_2$ is executed. If both outcomes are exceptions, the outcome of the TRY is the exception from $S_2$.

## 3.9 Loop

A statement of the form:

```
LOOP S END
```

repeatedly executes S until it raises the exit-exception. Informally it is like:

```
TRY S; S; S; ...
EXCEPT exit-exception => (*skip*)
END
```

## 3.10 Exit

The statement

```
EXIT
```

raises the exit-exception. An EXIT statement must be textually enclosed by a LOOP, WHILE, REPEAT, or FOR statement.

We define EXIT and RETURN in terms of exceptions in order to specify their interaction with the exception handling statements. As a pathological example, consider the following code, which is an elaborate infinite loop:

```
LOOP
  TRY
    TRY EXIT FINALLY RAISE E END
  EXCEPT
    E => (*skip*)
  END
END
```

## 3.11 Return

A RETURN statement for a proper procedure has the form:

```
RETURN
```

The statement raises the return-exception with no argument. It is allowed only in the body of a proper procedure.

[2] Some programmers use the semicolon as a statement terminator, some as a statement separator. Similarly, some use the vertical bar in case statements as a case initiator, some as a separator. Modula–3 allows both styles. This report uses both operators as separators.

A RETURN statement for a function procedure has the form:

```
RETURN Expr
```

where Expr is an expression assignable to the result type of the procedure. The statement raises the return-exception with the argument Expr. It is allowed only in the body of a function procedure.

Failure to return a value from a function procedure is a checked runtime error.

The effect of raising the return exception is to terminate the current procedure activation. To be precise, a call on a proper procedure with body B is equivalent (after binding the arguments) to:

```
TRY B EXCEPT
  return-exception => (*skip*) 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
```

## 3.12 If

An IF statement has the form:

```
IF     B1 THEN S1
ELSIF  B2 THEN S2
 ...
ELSIF  Bn THEN Sn
ELSE S0
END
```

where the B's are boolean expressions and the S's are statements. The "ELSE $S_0$" and each "ELSIF $B_i$ THEN $S_i$" are optional.

The statement evaluates the B's in order until some $B_i$ evaluates to TRUE, and then executes $S_i$. If none of the expressions evaluates to TRUE and "ELSE $S_0$" is present, $S_0$ is executed. If none of the expressions evaluates to TRUE and "ELSE $S_0$" is absent, the statement is a no-op (except for any side-effects of the B's).

## 3.13 While

If B is an expression of type BOOLEAN and S is a statement:

```
WHILE B DO S END
```

is shorthand for:

```
LOOP IF B THEN S ELSE EXIT END END
```

## 3.14 Repeat

If B is an expression of type BOOLEAN and S is a statement:

```
REPEAT S UNTIL B
```

is shorthand for:

```
LOOP S; IF B THEN EXIT END END
```

## 3.15 With

A WITH statement has the form:

```
WITH id = e DO S END
```

where id is an identifier, e an expression, and S a statement. The statement declares id with scope S as an alias for the variable e or as a readonly name for the value e. The expression e is evaluated once, at entry to the WITH statement.

The statement is like the procedure call P(e), where P is declared as:

```
PROCEDURE P(mode id: type of e) =
  BEGIN S END P;
```

If e is a writable designator, mode is VAR; otherwise, mode is READONLY. (Section 6.3 explains designators.) The only difference between the WITH statement and the call P(e) is that free variables, RETURNs, and EXITs that occur in the WITH statement are interpreted in the context of the WITH statement, not in the context of P.

A single WITH can contain multiple bindings, which are evaluated sequentially. That is: WITH $id_1$ = $e_1$, $id_2$ = $e_2$, ... DO is equivalent to: WITH $id_1$ = $e_1$ DO WITH $id_2$ = $e_2$ DO ....

## 3.16 For

A FOR statement has the form:

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

where id is an identifier, first and last are ordinal expressions with the same base type, step is an integer-valued expression, and S is a statement. "BY step" is optional; if omitted, step defaults to 1.

The identifier id denotes a readonly variable whose scope is S and whose type is the common basetype of first and last.

If id is an integer, the statement steps id through the values first, first+step, first+2*step, ..., stopping when the value of id passes last. S executes once for each value; if the sequence of values is empty, S never executes. The expressions first, last, and step are evaluated once, before the loop is entered. If step is negative, the loop iterates downward.

The case in which id is an element of an enumeration is similar. In either case, the semantics are defined precisely by the following rewriting, in which T is the type of id and in which i, done, and delta stand for variables that do not occur in the FOR statement:

```
VAR
  i := ORD(first);
  done := ORD(last); delta := step;
BEGIN
  IF delta >= 0 THEN
    WHILE i <= done DO
      WITH id = VAL(i, T) DO S END;
      INC(i, delta)
    END
  ELSE
    WHILE i >= done DO
      WITH id = VAL(i, T) DO S END;
      INC(i, delta)
    END
  END
END
```

If the upper bound of the loop is `LAST(INTEGER)`, it should be rewritten as a `WHILE` loop to avoid overflow.

## 3.17 Case

A `CASE` statement has the form:

```
CASE Expr OF
  L1 => S1
| ...
| Ln => Sn
ELSE S0
END
```

where `Expr` is an expression whose type is an ordinal type and each `L` is a list of constant expressions or ranges of constant expressions denoted by "$e_1 .. e_2$", which represent the values from $e_1$ to $e_2$ inclusive. If $e_1$ exceeds $e_2$, the range is empty. It is a static error if the sets represented by any two `L`'s overlap or if the value of any of the constant expressions is not a member of the type of `Expr`. The "`ELSE` $S_0$" is optional.

The statement evaluates `Expr`. If the resulting value is in any $L_i$, then $S_i$ is executed. If the value is in no $L_i$ and "`ELSE` $S_0$" is present, then it is executed. If the value is in no $L_i$ and "`ELSE` $S_0$" is absent, a checked runtime error occurs.

## 3.18 Typecase

A `TYPECASE` statement has the form:

```
TYPECASE Expr OF
  T1 (v1) => S1
| ...
| Tn (vn) => Sn
ELSE S0
END
```

where `Expr` is an expression whose type is a reference type, the `S`'s are statements, the `T`'s are reference types, and the `v`'s are identifiers. It is a static error if `Expr` has type `ADDRESS` or if any `T` is not a subtype of the type of `Expr`. The "`ELSE` $S_0$" and each "$(v_i)$" are optional.

The statement evaluates `Expr`. If the resulting reference value is a member of any listed type $T_i$, then $S_i$ is executed, for the minimum such $i$. (Thus a `NULL` case is useful only if it comes first.) If the value is a member of no listed type and "`ELSE` $S_0$" is present, then it is executed. If the value is a member of no listed type and "`ELSE` $S_0$" is absent, a checked runtime error occurs.

Each $(v_i)$ declares a variable whose type is $T_i$ and whose scope is $S_i$. If $v_i$ is present, it is initialized to the value of `Expr` before $S_i$ is executed.

If $(v_i)$ is absent, then $T_i$ can be a list of type expressions separated by commas, as shorthand for a list in which the rest of the branch is repeated for each type expression. That is:

```
T1, ..., Tn => S
```

is shorthand for:

```
T1 => S | ... | Tn => S
```

For example:

```
PROCEDURE ToText(r: REFANY): TEXT =
    (* Assume r = NIL or r^ is a
       BOOLEAN or INTEGER. *)
  BEGIN
    TYPECASE r OF
      NULL => RETURN "NIL"
    | REF BOOLEAN (rb) =>
        RETURN Fmt.Bool(rb^)
    | REF INTEGER (ri) =>
        RETURN Fmt.Int(ri^)
    END
  END ToText;
```

## 3.19 Lock

A `LOCK` statement has the form:

```
LOCK mu DO S END
```

where `S` is a statement and `mu` is an expression. It is equivalent to:

```
WITH m = mu DO
  Thread.Acquire(m);
  TRY S FINALLY
    Thread.Release(m) END
END
```

where `m` stands for a variable that does not occur in `S`. (The `Thread` interface is presented in Section 9.2.)

## 3.20 Inc and Dec

`INC` and `DEC` statements have the form:

```
INC(v, n)
DEC(v, n)
```

where v designates a variable of an ordinal type[1] and n is an optional integer-valued argument. If omitted, n defaults to 1. The statements increment and decrement v by n, respectively. The statements are equivalent to:

```
WITH x = v DO
  x := VAL(ORD(x) + n, T) END
WITH x = v DO
  x := VAL(ORD(x) - n, T) END
```

where T is the type of v and x stands for a variable that does not appear in n. As a consequence, the statements check for range errors.

## 4 Declarations

*There are two basic methods of declaring high or low before the showdown in all High-Low Poker games. They are (1) simultaneous declarations, and (2) consecutive declarations.... It is a sad but true fact that the consecutive method spoils the game.*
*—John Scarne's Guide to Modern Poker*

A declaration introduces a name for a constant, type, variable, exception, or procedure. The scope of the name is the block containing the declaration. A block has the form:

```
Decls BEGIN S END
```

where Decls is a sequence of declarations and S is a statement, the executable part of the block. A block can appear as a statement or as the body of a module or procedure. The declarations of a block can introduce a name at most once, though a name can be redeclared in nested blocks, and a procedure declared in an interface can be redeclared in a module exporting the interface (Section 5). The order of declarations in a block does not matter, except to determine the order of initialization of variables.

### 4.1 Types

If T is an identifier and U a type (or type expression, since a type expression is allowed wherever a type is required), then:

```
TYPE T = U
```

declares T to be the type U.

### 4.2 Constants

If id is an identifier, T a type, and C a constant expression, then:

```
CONST id: T = C
```

[1] In unsafe modules, INC and DEC are extended to ADDRESS.

declares id as a constant with the type T and the value of C. The ": T" can be omitted, in which case the type of id is the type of C. If T is present it must contain C.

### 4.3 Variables

If id is an identifier, T a non-empty type other than an open array type, and E an expression, then:

```
VAR id: T := E
```

declares id as a variable of type T whose initial value is the value of E. Either ":= E" or ": T" can be omitted, but not both. If T is omitted, it is taken to be the type of E. If E is omitted, the initial value is an arbitrary value of type T. If both are present, E must be assignable to T.

The initial value is a shorthand that is equivalent to inserting the assignment id := E at the beginning of the executable part of the block. If several variables have initial values, their assignments are inserted in the order they are declared. For example:

```
VAR i: [0..5] := j; j: [0..5] := i;
BEGIN S END
```

initializes i and j to the same arbitrary value in [0..5]; it is equivalent to:

```
VAR i: [0..5]; j: [0..5];
BEGIN i := j; j := i; S END
```

If a sequence of identifiers share the same type and initial value, id can be a list of identifiers separated by commas. Such a list is shorthand for a list in which the type and initial value are repeated for each identifier. That is:

VAR $v_1$, ..., $v_n$: T := E

is shorthand for:

VAR $v_1$: T := E; ...; VAR $v_n$: T := E

This means that E is evaluated n times.

### 4.4 Procedures

There are two forms of procedure declaration:

```
PROCEDURE id sig = B id
PROCEDURE id sig
```

where id is an identifier, sig is a procedure signature, and B is a block. In both cases, the type of id is the procedure type determined by sig. The first form is allowed only in modules; the second form is allowed only in interfaces.

The first form declares id as a procedure constant whose signature is sig, whose body is B, and whose environment is the scope containing the declaration. The parameter names are treated as if they were declared at the outer level of B; the parameter types and default values are evaluated in the scope containing the procedure declaration. The procedure name `id` must be repeated after the `END` that terminates the body.

The second form declares `id` to be a procedure constant whose signature is `sig`. The procedure body is specified in a module exporting the interface, by a declaration of the first form.

## 4.5 Exceptions

If `id` is an identifier and `T` a type other than an open array type, then:

```
EXCEPTION id(T)
```

declares `id` as an exception with argument type `T`. If "`(T)`" is omitted, the exception takes no argument. An exception declaration is allowed only in an interface or in the outermost scope of a module. All declared exceptions are distinct.

## 4.6 Opaque types

An *opaque type* is a name that denotes an unknown subtype of some given reference type. For example, an opaque subtype of `REFANY` is an unknown traced reference type; an opaque subtype of `UNTRACED ROOT` is an unknown untraced object type. The actual type denoted by an opaque type name is called its *concrete type.*

Different scopes can reveal different information about an opaque type. For example, what is known in one scope only to be a subtype of `REFANY` could be known in another scope to be a subtype of `ROOT`.

An opaque type declaration has the form:

```
TYPE T <: U
```

where `T` is an identifier and `U` an expression denoting a reference type. It introduces the name `T` as an opaque type and reveals that `U` is a supertype of `T`. The concrete type of `T` must be revealed elsewhere in the program.

## 4.7 Revelations

A *revelation* introduces information about an opaque type into a scope. Unlike other declarations, revelations introduce no new names.

There are two kinds of revelations, *partial* and *complete*. A program can contain any number of partial revelations for an opaque type; it must contain exactly one complete revelation.

A partial revelation has the form:

```
REVEAL T <: V
```

where `V` is a type expression (possibly just a name) and `T` is an identifier (possibly qualified, as in Section 5.1) declared as an opaque type. It reveals that `V` is a supertype of `T`.

In any scope, the revealed supertypes of an opaque type must be linearly ordered by the subtype relation. That is, if it is revealed that `T <: U1` and `T <: U2`, it must also be revealed either that `U1 <: U2` or that `U2 <: U1`.

A complete revelation has the form:

```
REVEAL T = V
```

where `V` is a type expression (not just a name) whose outermost type constructor is a branded reference or object type and `T` is an identifier (possibly qualified) that has been declared as an opaque type. The revelation specifies that `V` is the concrete type for `T`. It is a static error if any type revealed in any scope as a supertype of `T` is not a supertype of `V`. Generally this error is detected at link time.

Distinct opaque types have distinct concrete types, since `V` includes a brand and all brands in a program are distinct.

A revelation is allowed only in an interface or in the outermost scope of a module. A revelation in an interface can be imported into any scope where it is required, as illustrated by the stack example in Section 5.4.

For example, consider:

```
INTERFACE I; TYPE T <: ROOT;
PROCEDURE P(x: T): T; END I.

INTERFACE IClass; IMPORT I;
REVEAL I.T <: MUTEX; END IClass.

INTERFACE IRep; IMPORT I;
  REVEAL I.T = MUTEX BRANDED OBJECT
    count: INTEGER END;
END IRep.
```

An importer of `I` sees `I.T` as an opaque subtype of `ROOT`, and is limited to allocating objects of type `I.T`, passing them to `I.P`, or declaring subtypes of `I.T`. An importer of `IClass` sees that every `I.T` is a `MUTEX`, and can therefore lock objects of type `I.T`. Finally, an importer of `IRep` sees the concrete type, and can access the `count` field.

## 4.8 Recursive declarations

A constant, type, or procedure declaration `N = E`, a variable declaration `N : E`, an exception declaration `N(E)`, or a revelation `N = E` is *recursive* if `N` occurs in any partial expansion of `E`. A variable declaration `N := I` where the type is omitted is recursive if `N` occurs in any partial expansion of the type `E` of `I`. Such declarations are allowed if every occurrence of `N` in any partial expansion of `E` is (1) within some occurrence of the type constructor `REF` or `PROCEDURE`, (2) within a field or method type of the type constructor `OBJECT`, or (3) within a procedure body.

Examples of legal recursive declarations:

```
TYPE
  List = REF RECORD
    x: REAL; link: List END;
  T = PROCEDURE(n: INTEGER; p: T);
  XList = X OBJECT link: XList END;
CONST N =
  BYTESIZE(REF ARRAY[0..N] OF REAL);
PROCEDURE P(b: BOOLEAN) =
  BEGIN IF b THEN P(NOT b) END END P;
```

```
    EXCEPTION E(PROCEDURE() RAISES {E});
    VAR v: REF ARRAY [0..BYTESIZE(v)]
      OF INTEGER;
```

Examples of illegal recursive declarations:

```
    TYPE
      T = RECORD x: T END;
      U = OBJECT METHODS m() := U.m END;
    CONST N = N+1;
    REVEAL I.T = I.T BRANDED OBJECT END;
    VAR v := P(); PROCEDURE P():
      ARRAY [0..LAST(v)] OF T;
```

Examples of legal non-recursive declarations:

```
    VAR n := BITSIZE(n);
    REVEAL T <: T;
```

## 5 Modules and interfaces

*Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what detail one can do without and yet preserve the spirit of the whole.*
*—Willa Cather*

A *module* is like a block, except for the visibility of names. An entity is visible in a block if it is declared in the block or in some enclosing block; an entity is visible in a module if it is declared in the module or in an interface that is imported or exported by the module.

An *interface* is a group of declarations. Declarations in interfaces are the same as in blocks, except that any variable initializations must be constant, and procedure declarations must specify only the signature, not the body.

A module `X` *exports* an interface `I` to supply bodies for one or more of the procedures declared in the interface. A module or interface `X` *imports* an interface `I` to make the entities declared in `I` visible in `X`.

A *program* is a collection of modules and interfaces that contains every interface imported or exported by any of its modules or interfaces, and in which no procedure, module, or interface is multiply defined. The effect of executing a program is to execute the bodies of each of its modules. The order of execution of the modules is constrained by the initialization rule in Section 5.6.

The module whose body is executed last is called the *main module*. Implementations are expected to provide a way to specify the main module, in case the initialization rule does not determine it uniquely. The recommended rule is that the main module be the one that exports the interface `Main`, whose contents are implementation-dependent.

Program execution terminates when the body of the main module terminates, even if concurrent threads of control are still executing.

The names of the modules and interfaces of a program are called *global* names. The method for looking up global names—for example, by file system search paths—is implementation-dependent.

### 5.1 Import statements

There are two forms of import statements. All imports of both forms are interpreted simultaneously: their order doesn't matter.

The first form is

```
    IMPORT I AS J
```

which imports the interface whose global name is `I` and gives it the local name `J`. The entities and revelations declared in `I` become accessible in the importing module or interface, but the entities and revelations imported into `I` do not. To refer to the entity declared with name `N` in the interface `I`, the importer must use the *qualified identifier* `J.N`.

The statement `IMPORT I` is short for `IMPORT I AS I`.

The second form is

```
    FROM I IMPORT N
```

which introduces `N` as the local name for the entity declared as `N` in the interface `I`. A local binding for `I` takes precedence over a global binding. For example,

```
    IMPORT I AS J, J AS I;
    FROM I IMPORT N
```

simultaneously introduces local names `J`, `I`, and `N` for the entities whose global names are `I`, `J`, and `J.N`, respectively.

It is illegal to use the same local name twice:

```
    IMPORT J AS I, K AS I;
```

is a static error, even if `J` and `K` are the same.

### 5.2 Interfaces

An interface has the form:

```
    INTERFACE id; Imports; Decls END id.
```

where `id` is an identifier that names the interface, `Imports` is a sequence of import statements, and `Decls` is a sequence of declarations that contains no procedure bodies or non-constant variable initializations. The names declared in `Decls` and the visible imported names must be distinct. It is a static error for two or more interfaces to form an import cycle.

### 5.3 Modules

A module has the form:

```
    MODULE id EXPORTS Interfaces;
    Imports;
    Block id.
```

where `id` is an identifier that names the module, `Interfaces` is a list of distinct names of interfaces exported by the module, `Imports` is a list of import statements, and `Block` is a block, the *body* of the module. The name `id` must be repeated after the END that terminates the body. "EXPORTS Interfaces" can be omitted, in which case Interfaces defaults to id.

If module M exports interface I, then all declared names in I are visible without qualification in M. Any procedure declared in I can be redeclared in M, with a body. The signature in M must be covered by the signature in I (as defined in Section 2.8). To determine the interpretation of keyword bindings in calls to the procedure, the signature in M is used within M; the signature in I is used everywhere else.

Except for the redeclaration of exported procedures, the names declared at the top level of Block, the visible imported names, and the names declared in the exported interfaces must be distinct.

For example, the following is illegal, since two names in exported interfaces coincide:

```
INTERFACE I; PROCEDURE X(); END I.

INTERFACE J; PROCEDURE X(); END J.

MODULE M EXPORTS I, J;
  PROCEDURE X() = ...;
END M.
```

The following is also illegal, since the visible imported name X coincides with the top-level name X:

```
INTERFACE I; PROCEDURE X(); END I.

MODULE M EXPORTS I;
  FROM I IMPORT X;
  PROCEDURE X() = ...;
END M.
```

But the following is legal, although peculiar:

```
INTERFACE I; PROCEDURE X(...);END I.

MODULE M EXPORTS I;
  IMPORT I;
  PROCEDURE X(...) = ...;
END M.
```

since the only visible imported name is I, and the coincidence between X as a top-level name and X as a name in an exported interface is allowed, assuming the interface signature covers the module signature. Within M, the interface declaration determines the signature of I.X and the module declaration determines the signature of X.

## 5.4 Example module and interface

Here is the canonical example of a public stack with hidden representation:

```
INTERFACE Stack;
  TYPE T <: REFANY;
  PROCEDURE Create(): T;
  PROCEDURE Push(VAR s: T; x: REAL);
  PROCEDURE Pop(VAR s: T): REAL;
END Stack.
```

```
MODULE Stack;
REVEAL T = BRANDED OBJECT
  item: REAL; link: T END;
PROCEDURE Create(): T =
  BEGIN RETURN NIL END Create;
PROCEDURE Push(VAR s: T; x: REAL) =
  BEGIN
    s := NEW(T, item:= x, link:= s)
  END Push;

PROCEDURE Pop(VAR s: T): REAL =
  VAR res: REAL;
  BEGIN
    res := s.item; s := s.link;
    RETURN res
  END Pop;
BEGIN
END Stack.
```

If the representation of stacks is required in more than one module, it should be moved to a private interface, so that it can be imported wherever it is required:

```
INTERFACE Stack (*...as before...*)
END Stack.

INTERFACE StackRep;
IMPORT Stack;
REVEAL Stack.T = BRANDED OBJECT
  item: REAL; link: Stack.T END
END StackRep.

MODULE Stack; IMPORT StackRep;
  (* Push, Pop & Create as before *)
BEGIN
END Stack.
```

## 5.5 Generics

In a generic interface or module, some of the imported interface names are treated as formal parameters, to be bound to actual interfaces when the generic is instantiated.

A generic interface has the form

```
GENERIC INTERFACE G(F1, ..., Fn);
  Body
END G.
```

where G is an identifier that names the generic interface, $F_1$, ..., $F_n$ is a list of identifiers, called the formal imports of G, and Body is a sequence of imports followed by a sequence of declarations, exactly as in a non-generic interface.

An instance of G has the form

```
INTERFACE I = G(A1, ..., An) END I.
```

where I is the name of the instance and $A_1$, ..., $A_n$ is a list of actual interfaces to which the formal imports of G are bound. The instance I is equivalent to an ordinary interface defined as follows:

```
INTERFACE I;
  IMPORT A1 AS F1, ..., An AS Fn;
  Body
END I.
```

A generic module has the form

```
GENERIC MODULE G(F1, ..., Fn);
  Body
END G.
```

where `G` is an identifier that names the generic module, $F_1$, ..., $F_n$ is a list of identifiers, called the formal imports of `G`, and `Body` is a sequence of imports followed by a block, exactly as in a non-generic module.

An instance of `G` has the form

```
MODULE I EXPORTS E = G(A1, ..., An)
END I.
```

where `I` is the name of the instance, `E` is a list of interfaces exported by `I`, and $A_1$, ..., $A_n$ is a list of actual interfaces to which the formal imports of `G` are bound. "`EXPORTS E`" can be omitted, in which case it defaults to "`EXPORTS I`". The instance `I` is equivalent to an ordinary module defined as follows:

```
MODULE I EXPORTS E;
  IMPORT A1 AS F1, ..., An AS Fn;
  Body
END I.
```

Notice that the generic module itself has no exports; they are supplied only when it is instantiated.

For example, here is a generic stack package:

```
GENERIC INTERFACE Stack(Elem);
  (*Elem.T not an open array type*)
  TYPE T <: REFANY;
  PROCEDURE Create(): T;
  PROCEDURE Push(VAR s:T; x:Elem.T);
  PROCEDURE Pop(VAR s:T): Elem.T;
END Stack.

GENERIC MODULE Stack(Elem);
REVEAL T = BRANDED OBJECT
  n: INTEGER; a: REF ARRAY OF Elem.T
END;
PROCEDURE Create(): T =
  BEGIN
    RETURN NEW(T, n:=0, a:=NIL)
  END Create;
PROCEDURE Push(VAR s:T; x:Elem.T) =
  BEGIN
    IF s.a = NIL THEN
      s.a :=
        NEW(REF ARRAY OF Elem.T, 5)
    ELSIF s.n > LAST(s.a^) THEN
      WITH temp =
        NEW(REF ARRAY OF Elem.T,
            2 * NUMBER(s.a^))
      DO
        FOR i := 0 TO LAST(s.a^) DO
          temp[i] := s.a[i]
        END;
        s.a := temp
      END
    END;
    s.a[s.n] := x;
    INC(s.n)
  END Push;

PROCEDURE Pop(VAR s: T): Elem.T =
  BEGIN
    DEC(s.n); RETURN s.a[s.n]
  END Pop;
BEGIN
END Stack.
```

To instantiate these generics to produce stacks of integers, add:

```
INTERFACE Integer; TYPE T = INTEGER;
END Integer.

INTERFACE IntStack = Stack(Integer)
END IntStack.

MODULE IntStack = Stack(Integer)
END IntStack.
```

Implementations are not expected to share code between different instances of a generic module, since this will not be possible in general.

Implementations are not required to typecheck uninstantiated generics, but they must typecheck their instances. For example, if one made the following mistake:

```
INTERFACE String;
  TYPE T = ARRAY OF CHAR;
END String.

INTERFACE StringStack = Stack(String)
END StringStack.

MODULE StringStack = Stack(String)
END StringStack.
```

everything would go well until the last line, when the compiler would attempt to compile a version of `Stack` in which the element type was an open array. It would then complain that the `NEW` call in `Push` does not have enough parameters.

## 5.6 Initialization

The order of execution of the modules in a program is constrained by the following rule: If module `M` depends on module `N` and `N` does not depend on `M`, then `N`'s body will be executed before `M`'s body, where:

- A module `M` *depends on* a module `N` if `M` uses an interface that `N` exports or if `M` depends on a module that depends on `N`.
- A module `M` *uses* an interface `I` if `M` imports or exports `I` or if `M` uses an interface that imports `I`.

Except for this constraint, the order of execution is implementation-dependent.

### 5.7 Safety

The keyword `UNSAFE` can precede the declaration of any interface or module to indicate that it is *unsafe*; that is, uses the unsafe features of the language (Section 7). An interface or module not explicitly labeled `UNSAFE` is called *safe*.

An interface is *intrinsically safe* if there is no way to produce an unchecked runtime error by using the interface in a safe module. If all modules that export a safe interface are safe, the compiler guarantees the intrinsic safety of the interface. If any of the modules that export a safe interface are unsafe, it is the programmer, rather than the compiler, who makes the guarantee.

It is a static error for a safe interface to import an unsafe one or for a safe module to import or export an unsafe interface.

# 6 Expressions

*The rules of logical syntax must follow of themselves,*
*if we only know how every single sign signifies.*
*—Ludwig Wittgenstein*

An expression prescribes a computation that produces a value or variable. Syntactically, an expression is either an operand, or an operation applied to arguments, which are themselves expressions. Operands are identifiers, literals, or types. An expression is evaluated by recursively evaluating its arguments and performing the operation. The order of argument evaluation is undefined for all operations except `AND` and `OR`.

### 6.1 Conventions for describing operations

To describe the argument and result types of operations, we use a notation like procedure signatures. But since most operations are too general to be described by a true procedure signature, we extend the notation in several ways.

The argument to an operation can be required to have a type in a particular class, such as an ordinal type, set type, etc. In this case the formal specifies a type class instead of a type. For example:

```
ORD      (x: Ordinal)        : INTEGER
```

The formal type `Any` specifies an argument of any type.

A single operation name can be overloaded, which means that it denotes more than one operation. In this case, we write a separate signature for each of the operations. For example:

```
ABS      (x: INTEGER)        : INTEGER
         (x: Float)          : Float
```

The particular operation will be selected so that each actual argument type is a subtype of the corresponding formal type or a member of the corresponding formal type class.

The argument to an operation can be an expression denoting a type. In this case, we write `Type` as the argument type. For example:

```
BYTESIZE (T: Type)           : CARDINAL
```

The result type of an operation can depend on its argument values (although the result type can always be determined statically). In this case, the expression for the result type contains the appropriate arguments. For example:

```
FIRST(T:FixedArrayType):IndexType(T)
```

`IndexType(T)` denotes the index type of the array type `T` and `IndexType(a)` denotes the index type of the array `a`. The definitions of `ElemType(T)` and `ElemType(a)` are similar.

### 6.2 Operation syntax

The operators that have special syntax are classified and listed in order of decreasing binding power below:

| | |
|---|---|
| `x.a` | infix dot |
| `f(x) a[i] T{x}` | applicative `(`, `[`, `{` |
| `p^` | postfix `^` |
| `+ -` | prefix arithmetics |
| `* / DIV MOD` | infix arithmetics |
| `+ - &` | infix arithmetics |
| `= # < <= >= > IN` | infix relations |
| `NOT` | prefix `NOT` |
| `AND` | infix `AND` |
| `OR` | infix `OR` |

All infix operators are left associative. Parentheses can be used to override the precedence rules. Here are some examples of expressions together with their fully parenthesized forms:

| Expression | Treated as | Rule |
|---|---|---|
| `M.F(x)` | `(M.F)(x)` | dot before P() |
| `Q(x)^` | `(Q(x))^` | P() before ^ |
| `- p^` | `- (p^)` | ^ before prefix - |
| `- a * b` | `(- a) * b` | prefix - before * |
| `a * b - c` | `(a * b) - c` | * before infix - |
| `x IN s - t` | `x IN (s - t)` | infix - before IN |
| `NOT x IN s` | `NOT (x IN s)` | IN before NOT |
| `NOT p AND q` | `(NOT p) AND q` | NOT before AND |
| `A OR B AND C` | `A OR (B AND C)` | AND before OR |

Operators without special syntax are *procedural*. An application of a procedural operator has the form `op(args)`, where `op` is the operation and `args` is the list of argument expressions. For example, `MAX` and `MIN` are procedural operators.

## 6.3 Designators

An identifier is a writable designator if it is declared as a variable, is a `VAR` or `VALUE` parameter, is a local of a `TYPECASE` or `TRY EXCEPT` statement, or is a `WITH` local that is bound to a writable designator. An identifier is a readonly designator if it is a `READONLY` parameter, a local of a `FOR` statement, or a `WITH` local bound to a non-designator or readonly designator.

The only operations that produce designators are dereferencing, subscripting, selection, and `SUBARRAY`.[1] This section defines these operations and specifies the conditions under which they produce designators.

`r^` denotes the referent of `r`; this operation is called *dereferencing*. The expression `r^` is always a writable designator. It is a static error if the type of `r` is `REFANY`, `ADDRESS`, `NULL`, an object type, or an opaque type, and a checked runtime error if `r` is `NIL`. The type of `r^` is the referent type of `r`.

`a[i]` denotes the (`i + 1 - FIRST(a)`)th element of the array `a`. The expression `a[i]` is a designator if `a` is, and is writable if `a` is. The expression `i` must be assignable to the index type of `a`. The type of `a[i]` is the element type of `a`. An expression of the form `a[`$i_1$`, ..., `$i_n$`]` is shorthand for `a[`$i_1$`]...[`$i_n$`]`. If `a` is a reference to an array, then `a[i]` is shorthand for `a^[i]`.

`r.f, o.f, I.x, T.m, E.id`

If `r` denotes a record, `r.f` denotes its `f` field. In this case `r.f` is a designator if `r` is, and is writable if `r` is. The type of `r.f` is the declared type of the field.

If `r` is a reference to a record, then `r.f` is shorthand for `r^.f`.

If `o` denotes an object and `f` names a data field specified in the type of `o`, then `o.f` denotes that data field of `o`. In this case `o.f` is a writable designator whose type is the declared type of the field.

If `I` denotes an imported interface, then `I.x` denotes the entity named `x` in the interface `I`. In this case `I.x` is a designator if `x` is declared as a variable; such a designator is always writable.

If `T` is an object type and `m` is the name of one of `T`'s methods, then `T.m` denotes the `m` method of type `T`. In this case `T.m` is not a designator. Its type is the procedure type whose first argument has mode `VALUE` and type `T`, and whose remaining arguments are determined by the method declaration for `m` in `T`. The name of the first argument is unspecified; thus in calls to `T.m`, this argument must be given positionally, not by keyword. `T.m` is a procedure constant.

If `E` is an enumerated type, then `E.id` denotes its value named `id`. In this case `E.id` is not a designator. The type of `E.id` is `E`.

`SUBARRAY(a: Array; from, for: CARDINAL): ARRAY OF ElemType(a)`

`SUBARRAY` produces a subarray of `a`. It does not copy the array; it is a designator if `a` is, and is writable if `a` is. If `a` is a multidimensional array, `SUBARRAY` applies only to the top-level array.

The operation returns the subarray that skips the first `from` elements of `a` and contains the next `for` elements. Note that if `from` is zero, the subarray is a prefix of `a`, whether the type of `a` is zero-based or not. It is a checked runtime error if `from+for` exceeds `NUMBER(a)`.

Implementations may restrict or prohibit the `SUBARRAY` operation for arrays with packed element types.

## 6.4 Numeric literals

Numeric literals denote constant nonnegative integers or reals. The types of these literals are `INTEGER`, `REAL`, `LONGREAL`, and `EXTENDED`.

A literal `INTEGER` has the form `base_digits`, where `base` is one of "`2`", "`3`", ..., "`16`", and `digits` is a non-empty sequence of the decimal digits `0` through `9` plus the hexadecimal digits `A` through `F`. The "`base_`" can be omitted, in which case `base` defaults to `10`. The digits are interpreted in the given base. Each digit must be less than `base`. For example, `16_FF` and `255` are equivalent integer literals.

If no explicit base is present, the value of the literal must be at most `LAST(INTEGER)`. If an explicit base is present, the value of the literal must be less than $2^{\text{Word.Size}}$, and its interpretation uses the convention of the `Word` interface (Section 9.3). For example, on a sixteen-bit two's complement machine, `16_FFFF` and `-1` represent the same value.

A literal `REAL` has the form `decimal E exponent`, where `decimal` is a non-empty sequence of decimal digits followed by a decimal point followed by a non-empty sequence of decimal digits, and `exponent` is a non-empty sequence of decimal digits optionally beginning with a + or –. The literal denotes `decimal` times $10^{\text{exponent}}$. If "`E exponent`" is omitted, `exponent` defaults to `0`.

`LONGREAL` and `EXTENDED` literals are like `REAL` literals, but instead of `E` they use `D` and `X` respectively.

[1] In unsafe modules, `LOOPHOLE` can also produce a designator.

Case is not significant in digits, prefixes or scale factors. Embedded spaces are not allowed.

For example, `1.0` and `0.5` are valid, `1.` and `.5` are not; `6.624E-27` is a `REAL`, and `3.1415926535d0` a `LONGREAL`.

## 6.5 Text and character literals

A character literal is a pair of single quotes enclosing either a single ISO-Latin-1 printing character (excluding single quote) or an escape sequence. The type of a character literal is `CHAR`. A text literal is a pair of double quotes enclosing a sequence of ISO-Latin-1 printing characters (excluding double quote) and escape sequences. The type of a text literal is `TEXT`.

Here are the legal escape sequences and the characters they denote:

| | | | |
|---|---|---|---|
| `\n` | newline (linefeed) | `\f` | form feed |
| `\t` | tab | `\\` | backslash |
| `\r` | carriage return | `\"` | double quote |
| `\'` | single quote | `\`*nnn* | char with code `8_`*nnn* |

A \ followed by exactly three octal digits specifies the character whose code is that octal value. A \ that is not a part of one of these escape sequences is a static error.

For example, `'a'` and `'\''` are valid character literals, `'''` is not; `""` and `"Don't\n"` are valid text literals, `"""` is not.

## 6.6 Nil

The literal "`NIL`" denotes the value `NIL`. Its type is `NULL`.

## 6.7 Function application

A procedure call is an expression if the procedure returns a result. The type of the expression is the result type of the procedure.

## 6.8 Set, array, and record constructors

A set constructor has the form:

```
S{e1, ..., en}
```

where `S` is a set type and the `e`'s are expressions or ranges of the form `lo..hi`. The constructor denotes a value of type `S` containing the listed values and the values in the listed ranges. The `e`'s, `lo`'s, and `hi`'s must be assignable to the element type of `S`.

An array constructor has the form:

```
A{e1, ..., en}
```

where `A` is an array type and the `e`'s are expressions. The constructor denotes a value of type `A` containing the listed elements in the listed order. The `e`'s must be assignable to the element type of `A`. This means that if `A` is a multidimensional array, the `e`'s must themselves be array-valued expressions.

If `A` is a fixed array type and `n` is at least 1, then $e_n$ can be followed by "`, ..`" to indicate that the value of $e_n$ will be replicated as many times as necessary to fill out the array. It is a static error to provide too many or too few elements for a fixed array type.

A record constructor has the form:

```
R{Bindings}
```

where `R` is a record type and `Bindings` is a list of keyword or positional bindings, exactly as in a procedure call (Section 3.2). The list of bindings is rewritten to fit the list of fields and defaults of `R`, exactly as for a procedure call; the record field names play the role of the procedure formal parameters. The expression denotes a value of type `R` whose field values are specified by the rewritten binding.

The rewritten binding must bind only field names and must bind each field name exactly once. Each expression in the binding must be assignable to the type of the corresponding record field.

## 6.9 New

An allocation operation has the form:

```
NEW(T, ...)
```

where `T` is a reference type other than `REFANY`, `ADDRESS`, or `NULL`. The operation returns the address of a newly-allocated variable of `T`'s referent type; or if `T` is an object type, a newly-allocated data record paired with a method suite. The reference returned by `NEW` is distinct from all existing references. The allocated type of the new reference is `T`.

It is a static error if `T`'s referent type is empty. If `T` is declared as an opaque type, `NEW(T)` is legal only in scopes where `T`'s concrete type is known completely, or is known to be an object type.

The initial state of the referent generally represents an arbitrary value of its type. If `T` is an object type or a reference to a record or open array then `NEW` takes additional arguments to control the initial state of the new variable.

If `T` is a reference to an array with *k* open dimensions, the `NEW` operation has the form:

```
NEW(T, n1, ..., nk)
```

where the `n`'s are integer-valued expressions that specify the lengths of the new array in its first *k* dimensions. The values in the array will be arbitrary values of their If `T` is an object type or a reference to a record, the `NEW` operation has the form:

```
NEW(T, Bindings)
```

where `Bindings` is a list of keyword bindings used to initialize the new fields. Positional bindings are not allowed.

Each binding `f := v` initializes the field `f` to the value `v`. Fields for which no binding is supplied will be initialized to their defaults if they have defaults; otherwise they will be initialized to arbitrary values of their types.

If T is an object type then Bindings can also include method overrides of the form m := P, where m is a method of T and P is a top-level procedure constant. This is syntactic sugar for the allocation of a subtype of T that includes the given overrides, in the given order. For example, NEW(T, m := P) is sugar for NEW(T OBJECT OVERRIDES m := P END).

The order of the keyword bindings for the fields makes no difference.

## 6.10 Arithmetic operations

The basic arithmetic operations are built into the language; additional operations are provided by the required interfaces in Section 9.5.

To test or set the implementation's behavior for overflow, underflow, rounding, and division by zero, see the required interface FloatMode (Section 9.6). Modula–3 arithmetic was designed to support the IEEE floating-point standard, but not to require it.

To perform arithmetic operations modulo the word size, programs should use the routines in the required interface Word (Section 9.3).

Implementations must not rearrange the computation of expressions in a way that could affect the result. For example, (x+y)+z generally cannot be computed as x+(y+z), since addition is not associative either for bounded integers or for floating-point values.

```
prefix   +   (x: INTEGER)    : INTEGER
         +   (x: Float)      : Float

infix    +   (x,y: INTEGER)  : INTEGER
         +   (x,y: Float)    : Float
             (x,y: Set)      : Set
```

As a prefix operator, +x returns x. As an infix operator on numeric arguments, + denotes addition. On sets, + denotes set union. That is, e IN (x + y) if and only if (e IN x) OR (e IN y). The types of x and y must be the same, and the result is the same type as both. In unsafe modules, + is extended to ADDRESS.

```
prefix   -   (x: INTEGER)    : INTEGER
             (x: Float)      : Float

infix    -   (x,y: INTEGER)  : INTEGER
             (x,y: Float)    : Float
             (x,y: Set)      : Set
```

As a prefix operator, -x is the negative of x. As an infix operator on numeric arguments, - denotes subtraction. On sets, - denotes set difference. That is, e IN (x - y) if and only if (e IN x) AND NOT (e IN y). The types of x and y must be the same, and the result is the same type as both. In unsafe modules, - is extended to ADDRESS.

```
infix  , *   (x,y: INTEGER)  : INTEGER
             (x,y: Float)    : Float
             (x,y: Set)      : Set }
```

On numeric arguments, * denotes multiplication. On sets, * denotes intersection. That is, e IN (x * y) if and only if (e IN x) AND (e IN y). The types of x and y must be the same, and the result is the same type as both.

```
infix     /   (x,y: Float)    : Float
              (x,y: Set)      : Set
```

On reals, / denotes division. On sets, / denotes symmetric difference. That is, e IN (x / y) if and only if (e IN x) # (e IN y). The types of x and y must be the same, and the result is the same type as both.

```
infix     DIV  (x,y: INTEGER): INTEGER

infix     MOD  (x,y: INTEGER): INTEGER
          MOD  (x, y: Float) : Float
```

The value x DIV y is the floor of the quotient of x and y; that is, the maximum integer not exceeding the real number z such that z * y = x. For integers x and y, the value of x MOD y is defined to be x - y * (x DIV y).

This means that for positive y, the value of x MOD y lies in the interval [0 .. y-1], regardless of the sign of x. For negative y, the value of x MOD y lies in the interval [y+1 .. 0], regardless of the sign of x.

If x and y are floats, the value of x MOD y is x - y * FLOOR(x / y). This may be computed as a Modula–3 expression, or by a method that avoids overflow if x is much greater than y. The types of x and y must be the same, and the result is the same type as both.

```
ABS      (x: INTEGER)        : INTEGER
         (x: Float)          : Float
```

ABS(x) is the absolute value of x. If x is a float, the type of ABS(x) is the same as the type of x.

```
FLOAT  (x:INTEGER;  T:Type := REAL): T
       (x:Float;    T:Type := REAL): T
```

FLOAT(x, T) is a floating-point value of type T that is equal to or very near x. The type T must be a floating-point type; it defaults to REAL. The exact semantics depend on the thread's current rounding mode, as defined in the required interface FloatMode (Section 9.6).

```
FLOOR    (x: Float)          : INTEGER
CEILING  (x: Float)          : INTEGER
```

FLOOR(x) is the greatest integer not exceeding x. CEILING(x) is the least integer not less than x.

```
ROUND    (r: Float)          : INTEGER
TRUNC    (r: Float)          : INTEGER
```

ROUND(r) is the nearest integer to r; ties are broken according to the constant RoundDefault in the required interface FloatMode (Section 9.6). TRUNC(r) rounds r toward zero; it equals FLOOR(r) for positive r and CEILING(r) for negative r.

```
MAX, MIN (x,y: Ordinal)     : Ordinal
         (x,y: Float)       : Float
```

MAX returns the greater of the two values x and y; MIN returns the lesser. If x and y are ordinals, they must have the same base type, which is the type of the result. If x and y are floats, they must have the same type, and the result is the same type as both.

## 6.11 Relations

```
infix      =, # (x, y: Any)  : BOOLEAN
```

The operator = returns TRUE if x and y are equal. The operator # returns TRUE if x and y are not equal. It is a static error if the type of x is not assignable to the type of y or vice versa.

Ordinals are equal if they have the same value. Floats are equal if the underlying implementation defines them to be; for example, on an IEEE implementation, +0 equals -0 and NaN does not equal itself. References are equal if they address the same location. Procedures are equal if they agree as closures; that is, if they refer to the same procedure body and environment. Sets are equal if they have the same elements. Arrays are equal if they have the same length and corresponding elements are equal. Records are equal if they have the same fields and corresponding fields are equal.

```
infix <=, >=  (x,y: Ordinal) : BOOLEAN
              (x,y: Float)   : BOOLEAN
              (x,y: ADDRESS) : BOOLEAN
              (x,y: Set)     : BOOLEAN
```

In the first three cases, <= returns TRUE if x is at most as large as y. In the last case, <= returns TRUE if every element of x is an element of y. In all cases, it is a static error if the type of x is not assignable to the type of y, or vice versa. The expression x >= y is equivalent to y <= x.

```
infix  >, <   (x,y: Ordinal) : BOOLEAN
              (x,y: Float)   : BOOLEAN
              (x,y: ADDRESS) : BOOLEAN
              (x,y: Set)     : BOOLEAN
```

In all cases, x < y means (x <= y) AND (x # y), an x > y means y < x. It is a static error if the type of x is not assignable to the type of y, or vice versa.

Warning: with IEEE floating-point, x <= y is not the same as NOT x > y.

```
infix  IN (e:Ordinal; s:Set) : BOOLEAN
```

Returns TRUE if e is an element of the set s. It is a static error if the type of e is not assignable to the element type of s. If the value of e is not a member of the element type, no error occurs, but IN returns FALSE.

## 6.12 Boolean operations

```
prefix   NOT(p:   BOOLEAN) : BOOLEAN
infix    AND(p,q: BOOLEAN) : BOOLEAN
infix    OR (p,q: BOOLEAN) : BOOLEAN
```

NOT p is the complement of p.

p AND q is TRUE if both p and q are TRUE. If p is FALSE, q is not evaluated.

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

## 6.13 Type operations

```
ISTYPE (x: Reference; T: RefType)
         : BOOLEAN
```

ISTYPE(x, T) is TRUE if and only if x is a member of T. T must be an object type or traced reference type, and x must be assignable to T.

```
NARROW(x: Reference; T: RefType) : T
```

NARROW(x, T) returns x after checking that x is a member of T. If the check fails, a runtime error occurs. T must be an object type or traced reference type, and x must be assignable to T.

```
TYPECODE(T: RefType)        : CARDINAL
        (r: REFANY)         : CARDINAL
        (r: UNTRACED ROOT)  : CARDINAL
```

Every object type or traced reference type (including NULL) has an associated integer code. Different types have different codes. The code for a type is constant for any single execution of a program, but may differ for different executions. TYPECODE(T) returns the code for the type T and TYPECODE(r) returns the code for the allocated type of r. It is a static error if T is REFANY or is not an object type or traced reference type.

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

ORD converts an element of an enumeration to the integer that represents its position in the enumeration order. The first value in any enumeration is represented by zero. If the type of element is a subrange of an enumeration T, the result is the position of the element within T, not within the subrange.

VAL is the inverse of ORD; it converts from a numeric position i into the element that occupies that position in an enumeration. If T is a subrange, VAL returns the element with the position i in the original enumeration type, not the subrange. It is a checked runtime error for the value of i to be out of range for T.

If *n* is an integer, ORD(*n*) = VAL(*n*, INTEGER) = *n*.

```
NUMBER   (T:OrdinalType)     : CARDINAL
         (A:FixedArrayType)  : CARDINAL
         (a:Array)           : CARDINAL
```

For an ordinal type T, NUMBER(T) returns the number of elements in T. For a fixed array type A, NUMBER(A) is defined by NUMBER(IndexType(A)). For an array a, NUMBER(a) is defined by NUMBER(IndexType(a)). In this case, the expression a will be evaluated only if it denotes an open array.

```
FIRST(T: OrdinalType) : BaseType(T)
     (T: FloatType) : T
     (A: FixedArrayType)
       : BaseType(IndexType(A))
     (a: Array)
       : BaseType(IndexType(a))

LAST (T: OrdinalType) : BaseType(T)
     (T: FloatType) : T
     (A: FixedArrayType)
       : BaseType(IndexType(A))
     (a: Array)
       : BaseType(IndexType(a))
```

For a non-empty ordinal type T, FIRST returns the smallest value of T and LAST returns the largest value. If T is the empty enumeration, FIRST(T) and LAST(T) are static errors. If T is any other empty ordinal type, the values returned are implementation-dependent, but they satisfy FIRST(T) > LAST(T).

For a floating-point type T, FIRST(T) and LAST(T) are the smallest and largest values of the type, respectively. On IEEE implementations, these are minus and plus infinity.

For a fixed array type A, FIRST(A) is defined by FIRST(IndexType(A)) and LAST(A) by LAST(IndexType(A)). Similarly, for an array a, FIRST(a) and LAST(a) are defined by FIRST(IndexType(a)) and LAST(IndexType(a)). The expression a will be evaluated only if it is an open array. Note that if a is an open array, FIRST(a) and LAST(a) have type INTEGER.

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

BYTESIZE(x: Any)          : CARDINAL
        (T: Type)         : CARDINAL

ADRSIZE (x: Any)          : CARDINAL
        (T: Type)         : CARDINAL
```

These operations return the size of the variable x or of variables of type T. BITSIZE returns the number of bits, BYTESIZE the number of 8-bit bytes, and ADRSIZE the number of addressable locations. In all cases, x must be a designator and T must not be an open array type. A designator x will be evaluated only if its type is an open array type.

### 6.14 Text operations

```
infix    &   (a,b: TEXT)    : TEXT
```

a & b is the concatenation of a and b, that is, the characters from a followed by the characters from b. The expression a & b is shorthand for Text.Cat(a, b) (Section 9.1).

### 6.15 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 statically. All operations are legal in constant expressions except for ADR, LOOPHOLE, TYPECODE, NARROW, ISTYPE, SUBARRAY, NEW, dereferencing (explicit or implicit), and the only procedures that can be applied are the functions in the Word interface (Section 9.3).

A variable can appear in a constant expression only as an argument to FIRST, LAST, NUMBER, BITSIZE, BYTESIZE, or ADRSIZE, and such a variable must not have an open array type. Literals and top-level procedure constants are legal in constant expressions.

## 7 Unsafe operations

*There are some cases that no law can be framed to cover.*
*—Aristotle*

The features defined in this section can potentially cause unchecked runtime errors and are thus forbidden in safe modules.

An unchecked type transfer operation has the form:

```
LOOPHOLE(e, T)
```

where e is an expression whose type is not an open array type and T is a type. It denotes e's bit pattern interpreted as a variable or value of type T. It is a designator if e is, and is writable if e is. An unchecked runtime error can occur if e's bit pattern is not a legal T, or if e is a designator and some legal bit pattern for T is not legal for e.

If T is not an open array type, BITSIZE(e) must equal BITSIZE(T). If T is an open array type, its element type must not be an open array type, and e's bit pattern is interpreted as an array whose length is BITSIZE(e) divided by BITSIZE(the element type of T). The division must come out even.

The following operations are primarily used for address arithmetic:

```
ADR (VAR x: Any)           : ADDRESS
+   (x:ADDRESS; y:INTEGER) : ADDRESS
-   (x:ADDRESS; y:INTEGER) : ADDRESS
-   (x,y: ADDRESS)         : INTEGER
```

ADR(x) is the address of the variable x. The actual argument must be a designator but need not be writable. The operations + and - treat addresses as integers. The validity of the addresses produced by these operations is implementation-dependent. For example, the address of a variable in a local procedure frame is probably valid only for the duration of the call. The address of the referent of a traced reference is probably valid only as long as traced references prevent it from being collected (and not even that long if the implementation uses a compacting collector).

In unsafe modules the INC and DEC statements apply to addresses as well as ordinals:

```
INC (VAR x:ADDRESS; n:INTEGER := 1)
DEC (VAR x:ADDRESS; n:INTEGER := 1)
```

These are short for x := x + n and x := x - n, except that x is evaluated only once.

A DISPOSE statement has the form:

```
DISPOSE (v)
```

where v is a writable designator whose type is not REFANY, ADDRESS, or NULL. If v is untraced, the statement frees the storage for v's referent and sets v to NIL. Freeing storage to which active references remain is an unchecked runtime error. If v is traced, the statement is equivalent to v := NIL. If v is NIL, the statement is a no-op.

In unsafe modules the definition of "assignable" for types is extended: two reference types T and U are assignable if T <: U or U <: T. The only effect of this change is to allow a value of type ADDRESS to be assigned to a variable of type UNTRACED REF T. It is an unchecked runtime error if the value does not address a variable of type T.

In unsafe modules the type constructor UNTRACED REF T is allowed for traced as well as untraced T, and the fields of untraced objects can be traced. If u is an untraced reference to a traced variable t, then the validity of the traced references in t is implementation-dependent, since the garbage collector probably will not trace them through u.

# 8 Syntax

*Care should be taken, when using colons and semicolons in the same sentence, that the reader understands how far the force of each sign carries.*
*—Robert Graves and Alan Hodge*

## 8.1 Keywords

| | | | |
|---|---|---|---|
| AND | EXCEPT | MODULE | SET |
| ANY | EXCEPTION | NOT | THEN |
| ARRAY | EXIT | OBJECT | TO |
| AS | EXPORTS | OF | TRY |
| BEGIN | FINALLY | OR | TYPE |
| BITS | FOR | OVERRIDES | TYPECASE |
| BRANDED | FROM | PROCEDURE | UNSAFE |
| BY | GENERIC | RAISE | UNTIL |
| CASE | IF | RAISES | UNTRACED |
| CONST | IMPORT | READONLY | VALUE |
| DIV | IN | RECORD | VAR |
| DO | INTERFACE | REF | WHILE |
| ELSE | LOCK | REPEAT | WITH |
| ELSIF | LOOP | RETURN | |
| END | METHODS | REVEAL | |
| EVAL | MOD | ROOT | |

## 8.2 Reserved Identifiers

These *reserved* identifiers cannot be redeclared:

| | | | |
|---|---|---|---|
| ABS | DISPOSE | LOOPHOLE | REFANY |
| ADDRESS | EXTENDED | MAX | ROUND |
| ADR | FALSE | MIN | SUBARRAY |
| ADRSIZE | FIRST | MUTEX | TEXT |
| BITSIZE | FLOAT | NARROW | TRUE |
| BOOLEAN | FLOOR | NEW | TRUNC |
| BYTESIZE | INC | NIL | TYPECODE |
| CARDINAL | INTEGER | NULL | VAL |
| CEILING | ISTYPE | NUMBER | |
| CHAR | LAST | ORD | |
| DEC | LONGREAL | REAL | |

## 8.3 Operators

The following characters and character pairs are classified as operators:

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

## 8.4 Comments

A comment is an arbitrary character sequence opened by (* and closed by *). Comments can be nested and can extend over more than one line.

## 8.5 Pragmas

A pragma is an arbitrary character sequence opened by <* and closed by *>. Pragmas can be nested and can extend over more than one line. Pragmas are hints to the implementation; they do not affect the language semantics.

We recommend supporting the two pragmas <*INLINE*> and <*EXTERNAL*>. The pragma <*INLINE*> precedes a procedure declaration to indicate that the procedure should be expanded at the point of call. The pragma <*EXTERNAL N:L*> precedes an interface or a declaration in an interface to indicate that the entity it precedes is implemented by the language L, where it has the name N. If ":L" is omitted, then the implementation's default external language is assumed. If "N" is omitted, then the external name is determined from the Modula–3 name in some implementation-dependent way.

## 8.6 Conventions for syntax

We use the following notation for defining syntax:

| | |
|---|---|
| X Y | X followed by Y |
| X \| Y | X or Y |
| [X] | X or empty |
| {X} | A possibly empty sequence of X's |
| X & Y | X or Y or X Y |

"Followed by" has greater binding power than `|` or `&`; parentheses are used to override this precedence rule. Nonterminals begin with an uppercase letter. Terminals are either keywords or quoted operators. The symbols `Ident`, `Number`, `TextLiteral`, and `Program` are defined in the token grammar in Section 8.12. Each production is terminated by a period. The syntax does not reflect the restrictions that revelations and exceptions can be declared only at the top level; nor does it include explicit productions for `NEW`, `INC`, and `DEC`, which parse like procedure calls.

## 8.7 Compilation unit productions

```
Compilation = [UNSAFE] (Interface | Module)
        | GenInf | GenMod.
Interface= INTERFACE Id ";" {Import} {Decl} END Id "."
         | INTERFACE Id "=" Id GenActls END Id ".".
Module   = MODULE Id [EXPORTS IdList] ";" {Import}
              Block Id "."
         | MODULE Id [EXPORTS IdList] "=" Id GenActls
              END Id ".".
GenInf   = GENERIC INTERFACE Id GenFmls ";" {Import}
           {Decl} END Id ".".
GenMod   = GENERIC MODULE Id GenFmls ";" {Import}
           Block Id ".".
Import   = AsImport | FromImport.
AsImport = IMPORT ImportItem {"," ImportItem} ";".
FromImport = FROM Id IMPORT IdList ";".
ImportItem = Id | Id AS Id.
GenFmls = "(" [IdList] ")".
GenActls = "(" [IdList] ")".
Block    = {Decl} BEGIN S END.

Decl     = CONST {ConstDecl ";"}
         | TYPE {TypeDecl ";"}
         | EXCEPTION {ExceptionDecl ";"}
         | VAR {VariableDecl ";"}
         | ProcedureHead ["=" Block Id] ";"
         | REVEAL {QualId ("=" | "<:") Type ";"}.
ConstDecl = Id [":" Type] "=" ConstExpr.
TypeDecl = Id ("=" | "<:") Type.
ExceptionDecl = Id ["(" Type ")"].
VariableDecl = IdList (":" Type & ":=" Expr).
ProcedureHead = PROCEDURE Id Signature.
Signature = "(" Formals ")" [":" Type] [RAISES Raises].
Formals = [ Formal {";" Formal} [";"] ].
Formal   = [Mode] IdList (":" Type & ":=" ConstExpr).
Mode     = VALUE | VAR | READONLY.
Raises   = "{" [ QualId {"," QualId} ] "}" | ANY.
```

## 8.8 Statement productions

```
Stmt     = AssignSt | Block | CallSt | CaseSt | ExitSt
         | EvalSt | ForSt | IfSt | LockSt | LoopSt
         | RaiseSt | RepeatSt | ReturnSt | TCaseSt
         | TryXptSt | TryFinSt | WhileSt | WithSt.
S        = [ Stmt {";" Stmt} [";"] ].
AssignSt= Expr ":=" Expr.
CallSt   = Expr "(" [Actual {"," Actual}] ")".
CaseSt   = CASE Expr OF [Case] {"|" Case} [ELSE S] END.
ExitSt   = EXIT.
EvalSt   = EVAL Expr.
ForSt    = FOR Id ":=" Expr TO Expr [BY Expr] DO S END.
IfSt     = IF Expr THEN S {ELSIF Expr THEN S}
              [ELSE S] END.
LockSt   = LOCK Expr DO S END.
LoopSt   = LOOP S END.
RaiseSt = RAISE QualId ["(" Expr ")"].
RepeatSt = REPEAT S UNTIL Expr.
ReturnSt = RETURN [Expr].
TCaseSt = TYPECASE Expr OF [TCase] {"|" TCase}
              [ELSE S] END.
TryXptSt = TRY S EXCEPT [Handler] {"|" Handler}
              [ELSE S] END.
TryFinSt = TRY S FINALLY S END.
WhileSt  = WHILE Expr DO S END.
WithSt   = WITH Binding {"," Binding} DO S END.
Case     = Labels {"," Labels} "=>" S.
Labels   = ConstExpr [".." ConstExpr].
Handler  = QualId {"," QualId} ["(" Id ")"] "=>" S.
TCase    = Type {"," Type} ["(" Id ")"] "=>" S.
Binding  = Id "=" Expr.
Actual   = Type | [Id ":="] Expr .
```

## 8.9 Type productions

```
Type     = TypeName | ArrayType | PackedType | EnumType
         | ObjectType | ProcedureType | RecordType
         | RefType | SetType | SubrangeType
         | "(" Type ")".
ArrayType = ARRAY [Type {"," Type}] OF Type.
PackedType = BITS ConstExpr FOR Type.
EnumType = "{" [IdList] "}".
ObjectType = [TypeName | ObjectType] [Brand] OBJECT
                Fields [METHODS Methods]
                [OVERRIDES Overrides] END.
ProcedureType = PROCEDURE Signature.
RecordType = RECORD Fields END.
RefType = [UNTRACED] [Brand] REF Type.
SetType = SET OF Type.
SubrangeType = "[" ConstExpr ".." ConstExpr "]".
Brand    = BRANDED [TextLiteral].
Fields   = [ Field {";" Field} [";"] ].
Field    = IdList (":" Type & ":=" ConstExpr).
Methods  = [ Method {";" Method} [";"] ].
Method   = Id Signature [":=" ConstExpr].
Overrides = [ Override {";" Override} [";"] ].
Override = Id ":=" ConstExpr .
```

## 8.10 Expression productions

```
ConstExpr = Expr.
Expr     = E1 {OR E1}.
E1       = E2 {AND E2}.
E2       = {NOT} E3.
E3       = E4 {Relop E4}.
E4       = E5 {Addop E5}.
E5       = E6 {Mulop E6}.
E6       = {"+" | "-"} E7.
E7       = E8 {Selector}.
E8       = Id | Number | CharLiteral | TextLiteral
         | Constructor | "(" Expr ")".
Relop    = "=" | "#" | "<" | "<=" | ">" | ">=" | IN.
```

```
Addop   = "+" | "-" | "&".
Mulop   = "*" | "/" | DIV | MOD.
Selector = "^" | "." Id | "[" Expr {"," Expr} "]"
         | "(" [ Actual {"," Actual} ] ")".
Constructor = Type "{" [ SetCons | RecordCons
                       | ArrayCons ] "}".
SetCons = SetElt {"," SetElt}.
SetElt = Expr [".." Expr].
RecordCons = RecordElt {"," RecordElt}.
RecordElt = [Id ":="] Expr.
ArrayCons = Expr {"," Expr} ["," ".."].
```

### 8.11 Miscellaneous productions

```
IdList   = Id {"," Id}.
QualId   = Id ["." Id].
TypeName = QualId | ROOT | UNTRACED ROOT.
```

### 8.12 Token productions

To read a token, first skip all blanks, tabs, newlines, carriage returns, vertical tabs, form feeds, comments, and pragmas. Then read the longest sequence of characters that forms an operator (as defined in Section 8.3) or an `Id` or `Literal`, as defined here.

An `Id` is a case-significant sequence of letters, digits, and underscores that begins with a letter. An `Id` is a keyword if it appears in Section 8.1, a reserved identifier if it appears in Section 8.2, and an ordinary identifier otherwise.

In the following grammar, terminals are characters surrounded by double-quotes and the special terminal `DQUOTE` represents double-quote itself.

```
Id      = Letter {Letter | Digit | "_"}.
Literal = Number | CharLiteral | TextLiteral.
CharLiteral = "'" (PrintingChar | Escape | DQUOTE) "'".
TextLiteral = DQUOTE {PrintingChar | Escape | "'"}
              DQUOTE.
Escape  = "\" "n" | "\" "t" | "\" "r" | "\" "f"
        | "\" "\" | "\" "'" | "\" DQUOTE
        | "\" OctalDigit OctalDigit OctalDigit.
Number  = Digit {Digit}
        | Digit {Digit} "_" HexDigit {HexDigit}
        | Digit {Digit} "." Digit {Digit} [Exp].
Exp     = ("E" | "e" | "D" | "d" | "X" | "x") ["+"
        | "-"] Digit {Digit}.
HexDigit= Digit | "A" | "B" | "C" | "D" | "E" | "F"
        | "a" | "b" | "c" | "d" | "e" | "f".
Digit   = "0" | "1" | ... | "9".
OctalDigit = "0" | "1" | ... | "7".
PrintingChar = Letter | Digit | OtherChar.
Letter  = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z".
OtherChar = " " | "!" | "#" | "$" | "%" | "&" | "(" | ")"
        | "*" | "+" | "," | "-" | "." | "/" | ":" | ";"
        | "<" | "=" | ">" | "?" | "@" | "[" | "]" | "^"
        | "_" | "`" | "{" | "|" | "}" | " "
        | ExtendedChar
ExtendedChar = any char with ISO-Latin-1 code in
               [8_240..8_377].
```

## 9 Standard Interfaces

The following fundamental interfaces must be provided by every Modula–3 implementation.

### 9.1 Text

Interface `Text` provides operations on text strings. Non-nil values of type `TEXT` represent zero-based sequences of characters. The value `NIL` does not represent any sequence of characters, it is not returned from any procedure in the interface, and it is a checked runtime error to pass it to any procedure in the interface.

```
INTERFACE Text;
TYPE T = TEXT;  (* Text.T is the same as TEXT *)
PROCEDURE Cat(t, u: T): T;
PROCEDURE Equal(t, u: T): BOOLEAN;
   (* TRUE if t and u have the same length and
   (case sensitive) contents. *)
PROCEDURE GetChar(t: T; i: CARDINAL): CHAR;
   (* Character i of t. A checked runtime error
   if i >= Length(t). *)
PROCEDURE Length(t: T): CARDINAL;
PROCEDURE Empty(t: T): BOOLEAN;
PROCEDURE Sub(t: T; start, length: CARDINAL): T;
   (* Return a subsequence of t: empty if
   start >= Length(t) or length = 0;
   otherwise the subsequence ranginf rom start to
   the minimum of start+length-1 and Length(t)-1. *)
PROCEDURE SetChars(VAR a: ARRAY OF CHAR; t: T);
   (* For each i from 0 to MIN(LAST(a), Length(t)-1),
   set a[i] to GetChar(t, i). *)
PROCEDURE FromChar(c: CHAR): T;
   (* A string containing the single character c. *)
PROCEDURE FromChars(READONLY a: ARRAY OF CHAR): T;
   (* A string containing the characters from a. *)
END Text.
```

### 9.2 Thread

Interface `Thread` provides synchronization primitives for multiple threads of control. A `Thread.T` is a handle on a thread. A `Mutex` is locked by some thread, or unlocked. A `Condition` is a set of waiting threads. A newly-allocated `Mutex` is unlocked; a newly-allocated `Condition` is empty. It is a checked runtime error to pass the `NIL` `Mutex`, `Condition`, or `T` to any procedure in the interface. The `Alerted` exception approximates asynchronous interrupts.

```
INTERFACE Thread;
TYPE
   T <: REFANY;
   Mutex = MUTEX;
   Condition <: ROOT;
   Closure = OBJECT  METHODS apply(): REFANY; END;
PROCEDURE Fork(cl: Closure): T;
   (* Handle on a new thread executing cl.apply() *)
PROCEDURE Join(t: T): REFANY;
   (* Wait for t to terminate and return its result *)
PROCEDURE Wait(m: Mutex; c: Condition);
   (* Calling thread must have m locked. Atomically
   unlocks m and waits on c. Then relocks m and
   returns. *)
PROCEDURE Acquire(m: Mutex);
   (* Wait until m is unlocked and then lock it. *)
PROCEDURE Release(m: Mutex);
```

```
   (* Calling thread must have m locked; unlocks m. *)
PROCEDURE Broadcast(c: Condition);
   (* All threads waiting on c become runnable. *)
PROCEDURE Signal(c: Condition);
   (* One or more threads waiting on c become
   runnable. *)
PROCEDURE Self(): T;
   (* Handle of the calling thread. *)
EXCEPTION Alerted;
PROCEDURE Alert(t: T);
   (* Mark t as an alerted thread. *)
PROCEDURE TestAlert(): BOOLEAN;
   (* TRUE if the calling thread has been
   marked alerted. *)
PROCEDURE AlertWait(m: Mutex; c: Condition)
   RAISES {Alerted};
   (* Like Wait, but if the thread is marked
   alerted at the time of the call or during the
   wait then m is locked and Alerted is raised. *)
PROCEDURE AlertJoin(t: T): REFANY RAISES {Alerted};
   (* Like Join, but if t is marked alerted at the
   time of call or sometime during the wait, then
   Alerted is raised. *)
CONST AtomicSize = ... (* memory-coherent block bits *);
END Thread.
```

## 9.3 Word

Interface Word provides operations on unsigned words. A value $w$ of type Word.T represents a sequence of $s$ (Word.Size) bits numbered right-to-left: $w_0, \ldots, w_{s-1}$. It also represents the unsigned number $\sum_{i=0}^{s-1} 2^i \times w_i$ .

```
INTERFACE Word;
TYPE T = INTEGER;
CONST Size = BITSIZE(T);
PROCEDURE Plus(x, y: T): T;   (* (x + y) MOD 2^Size *)
PROCEDURE Times(x, y: T): T;  (* (x * y) MOD 2^Size *)
PROCEDURE Minus(x, y: T): T;  (* (x - y) MOD 2^Size *)
PROCEDURE Divide(x, y: T): T;             (* x DIV y *)
PROCEDURE Mod(x, y: T): T;                (* x MOD y *)
PROCEDURE LT(x, y: T): BOOLEAN;   (* unsigned x < y *)
PROCEDURE LE(x, y: T): BOOLEAN;  (* unsigned x <= y *)
PROCEDURE GT(x, y: T): BOOLEAN;   (* unsigned x > y *)
PROCEDURE GE(x, y: T): BOOLEAN;  (* unsigned x >= y *)
PROCEDURE And(x, y: T): T;            (* bitwise AND *)
PROCEDURE Or(x, y: T): T;              (* bitwise OR *)
PROCEDURE Xor(x, y: T): T;            (* bitwise XOR *)
PROCEDURE Not(x: T): T;       (* bitwise complement *)
PROCEDURE Shift(x: T; n: INTEGER): T;
   (* Bitwise shift with 0 fill; left shift
   if n>0, right shift if n<0. *)
PROCEDURE Rotate(x: T; n: INTEGER): T;
   (* Bit i of the result equals bit
   (i - n) MOD Word.Size of x. *)
PROCEDURE Extract(x: T; i, n: CARDINAL): T;
   (* Take bits i through i+n-1 from x and return
   them as the least significant n bits of a
   word whose other bits are 0. *)
PROCEDURE Insert(x, y: T; i, n: CARDINAL): T;
   (* Return x with bits i through i+n-1 replaced
   by the least significant n bits of y. The other
   bits of x are unchanged. *)
END.
```

## 9.4 Real, LongReal, Extended

The interfaces Real, LongReal, and Extended each define a type T that is the corresponding built-in floating-point type. (Real.T is REAL, and so forth.) Each interface also defines five constants that characterize the floating-point numbers of that type:

- Base is the integer radix of the floating-point representation for T.
- Precision is the number of digits of precision (in the given base) for T.
- MaxFinite is the maximum finite value in T.
- MinPos is the minimum positive value in T.
- MinPosNormal is the minimum positive "normal" value in T; it differs from MinPos only if the representation has denormalized numbers (e.g., IEEE format).

Here is an example of Real that uses the IEEE single-precision, floating-point representation.

```
INTERFACE Real;
TYPE T = REAL;
CONST
   Base: INTEGER = 2;
   Precision: INTEGER = 24;
   MaxFinite: T = 3.40282347E+38;
   MinPos: T = 1.40239846E-45;
   MinPosNormal: T = 1.17549435E-38;
END Real.

INTERFACE LongReal; ... END LongReal.

INTERFACE Extended; ... END Extended.
```

## 9.5 Float, RealFloat, LongRealFloat, ExtendedFloat

Generic interface Float provides access to the floating-point operations required or recommended by the IEEE floating-point standard. Consult the standard for the precise specifications of the procedures. The comments here specify their effect when the arguments are ordinary numbers and no exception is raised. Implementations on non-IEEE machines that have values similar to NaNs and infinities should explain how these values behave in an implementation guide.

```
INTERFACE RealFloat = Float(Real) END RealFloat.
INTERFACE LongFloat = Float(LongReal) END LongFloat.
INTERFACE ExtendedFloat = Float(Extended)
   END ExtendedFloat.

GENERIC INTERFACE Float(Real);
TYPE T = Real.T;
PROCEDURE Scalb(x: T; n: INTEGER): T;         (* x*2^n *)
PROCEDURE Logb(x: T): T;       (* the exponent of x *)
PROCEDURE ILogb(x: T): INTEGER;
   (* Like Logb, but returns an integer, never
   raises an exception, and always returns
   n such that ABS(x) / Base^n is in [1..Base-1]
   even for denormalized numbers. *)
PROCEDURE NextAfter(x, y: T): T;
   (* Return the next neighbor of x in the direction
   towards y. If x = y, return x. *)
PROCEDURE CopySign(x, y: T): T;
   (* Return x with the sign of y. *)
```

```
PROCEDURE Finite(x: T): BOOLEAN;
   (* Return TRUE if x is strictly between
   -infinity and +infinity. *)
PROCEDURE IsNaN(x: T): BOOLEAN;
   (* Return FALSE if x represents a numerical
   (possibly infinite) value, and TRUE if x
   does not represent a numerical value. *)
PROCEDURE Sign(x: T): [0..1];      (* sign bit of x *)
PROCEDURE Differs(x, y: T): BOOLEAN;
   (* (x < y OR y < x) *)
PROCEDURE Unordered(x, y: T): BOOLEAN;
   (* NOT (x <= y OR y <= x) *)
PROCEDURE Sqrt(x: T): T;
   (* the square root of x *)
TYPE IEEEClass =
   {SignalingNaN, QuietNaN, Infinity,
    Normal, Denormal, Zero};
PROCEDURE Class(x: T): IEEEClass;
   (* Return the IEEE number class containing x. *)
END Float.
```

## 9.6 FloatMode

Interface `FloatMode` allows you to test the behavior of rounding and of numerical exceptions. On some implementations it also allows you to change the behavior, on a per-thread basis.

```
INTERFACE FloatMode;
CONST IEEE: BOOLEAN = ...;
              (* TRUE for full IEEE implementations. *)
EXCEPTION Failure;
TYPE RoundingMode =
   {MinusInfinity, PlusInfinity, Zero,
    Nearest, Vax, IBM370, Other};
CONST RoundDefault: RoundingMode = ...;
PROCEDURE SetRounding(md: RoundingMode)
   RAISES {Failure};
PROCEDURE GetRounding(): RoundingMode;

TYPE Flag =
   {Invalid, Inexact, Overflow, Underflow,
    DivByZero, IntOverflow, IntDivByZero};
CONST NoFlags = SET OF Flags {};
PROCEDURE GetFlags(): SET OF Flag;
        (* Return the flags for this thread *)
PROCEDURE SetFlags(s: SET OF Flag): SET OF Flag;
PROCEDURE ClearFlag(f: Flag);
EXCEPTION Trap(Flag);

TYPE Behavior = {Trap, SetFlag, Ignore};
PROCEDURE SetBehavior(f: Flag; b: Behavior)
   RAISES {Failure};
PROCEDURE GetBehavior(f: Flag): Behavior;
END FloatMode.
```