Modula-3 Report (revised)

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

Draft of 21 Oct 89

Contents

Preface

The goal of Modula-3 is to be as simple and safe as it can be while meeting the needs of modern systems programmers. Instead of exploring new features, we studied the features from the Modula family of languages that have proven themselves in practice and tried to simplify them and fit them into a harmonious language. We found that most of the successful features were aimed at one of two main goals: greater robustness, and a simpler, more systematic type system.

Modula-3 descends from Mesa[6], Modula-2[10], Cedar[3], and Modula-2+[8, 7]. It also resembles its cousins Object Pascal[5], Oberon[9], and Euclid[4].

Modula-3 retains one of Modula-2's most successful features, the provision for explicit interfaces between modules. It adds objects and classes, exception handling, garbage collection, lightweight processes (or *threads*), and the isolation of unsafe features.

The Modula-3 report was published by Olivetti and Digital in August 1988. Implementation efforts followed shortly at both companies. In January 1989, the committee revised the language to reflect the experiences of these implementation teams. The main changes were the introduction of branded reference types, the requirement that opaque types be branded, the legalization of opaque supertypes, and the new flexibility in revealing information about an opaque type.

Modula-3 Report

He that will not apply new remedies must expect new evils: for time is the greatest innovator, and if time of course alter things to the worse, and wisdom and counsel shall not alter them to the better, what shall be the end?
—Francis Bacon

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 allowed wherever a type is required.

In any scope, a type name is either *opaque* or *concrete*. An opaque type name denotes a reference type with an unknown referent type. A concrete type name denotes a type that is known in the scope (except for the referent types of opaque types that occur within it). The `REVEAL` statement can be used to make an opaque type name concrete within a scope by making its referent type known (Section 4, page 31).

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 that 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`.

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, page 27).

Ordinal types

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

$\texttt{TYPE T = \{id}_1\texttt{, id}_2\texttt{, ..., id}_n\texttt{\}}$

where the `id`'s are distinct identifiers. The type `T` is an ordered set of `n` values; the expression $\texttt{T.id}_i$ 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, page 48). 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, page 46).

Here are the predeclared ordinal types:

`INTEGER`	All integers represented by the implementation
`CARDINAL`	The 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. `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 form. 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 form of `T1` is an enumeration while the expanded form 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`.

Floating-point types

There are two built-in floating point types:

`REAL`	Contains all single-precision floating point values
`LONGREAL`	Contains all double-precision floating point values

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 variable with a fixed array type is determined at compile time. The length of a variable with an open array type is determined at runtime, when the variable is allocated or bound. It cannot be changed thereafter.

The *shape* of a multi-dimensional 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 (Section 3, page 17).

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 [0..2] OF REAL
VAR b: ARRAY [5..7] OF REAL
```

The variables `a` and `b` range over the same set of values, namely the set of real sequences of length three. But `a[i]` and `b[i]` are interpreted differently; for example, the first element of `a` is designated `a[0]` while the first element of `b` is designated `b[5]`. This interpretation is unchanged by the assignment `a := b`, since assignment changes a variable's value but not its type.

An expression of the form:

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

is shorthand for:

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

This shorthand is eliminated from the expanded form of the type.

An expression of the form $\texttt{a[i}_1\texttt{, ..., i}_n\texttt{]}$ is shorthand for $\texttt{a[i}_1\texttt{]...[i}_n\texttt{]}$.

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 set of an open array variable 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;
  Priority = {Background, Normal, High};
  ReadyQueue = ARRAY Priority OF Queue.T
```

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 (page 41) or allocated (page 42). 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:

$\mathtt{f}_1$**, ...,** $\mathtt{f}_m$**: Type := default**

is shorthand for:

$\mathtt{f}_1$**: Type := default; ...;** $\mathtt{f}_m$**: Type := default**

The shorthand is eliminated from the expanded form of the type. The default values are included in the expanded form.

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
```

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.

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.

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`.

References

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

A reference type is either *fixed*, *open*, or an object type. The members of a fixed reference type address variables of some fixed type; the members of an open reference type address variables of any type. An object type is intermediate between a fixed and open reference type: all variables addressed by a member of an object type share a common set of fields and methods. This section describes fixed and open reference types; Section 2, page 11, describes object types.

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.

There are exactly three open reference types (`NULL` is included by convention):

`REFANY`	Contains all traced references
`ADDRESS`	Contains all untraced references
`NULL`	Contains only `NIL`

The `TYPECASE` statement (Section 3, page 27) can be used to determine the referent type of a variable of type `REFANY` or of an object, but there is no such operation for variables of type `ADDRESS`.

A declaration for a fixed 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`.

Untraced reference types are similar, but with the restriction that it is unsafe for an untraced reference to point at a type that the garbage collector must trace. We thus extend the definition of "traced": a 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. Otherwise a type is *untraced*.

A declaration for a fixed 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 literal called the *brand*. Brands distinguish types

[1] This restriction is lifted in unsafe modules.

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`.

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-M3-040776" REF INTEGER;
  Apple = BRANDED REF INTEGER;
  Orange = BRANDED REF INTEGER;
```

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:

$$\texttt{(mode}_1\ \texttt{name}_1\texttt{: type}_1\ \texttt{:= default}_1\texttt{;}$$
$$\texttt{...;}$$
$$\texttt{mode}_n\ \texttt{name}_n\texttt{: type}_n\ \texttt{:= default}_n\texttt{): R RAISES \{S\}}$$

where:

- Each $\mathtt{mode}_i$ is a parameter mode, which can be **VALUE**, **VAR**, or **READONLY**. If $\mathtt{mode}_i$ is omitted, it defaults to **VALUE**.
- Each $\mathtt{name}_i$ is an identifier, the name of parameter `i`. The parameter names must be distinct.
- Each $\mathtt{type}_i$ is a type, the type of parameter `i`.
- Each $\mathtt{default}_i$ is a constant expression to be used as a default value for parameter `i`. If $\mathtt{mode}_i$ is **VAR**, it is a static error to include "$\mathtt{:=\ default}_i$". If $\mathtt{mode}_i$ is **READONLY** or **VALUE**, either "$\mathtt{:=\ default}_i$" or "$\mathtt{:\ type}_i$" can be omitted, but not both. If $\mathtt{default}_i$ is omitted, calls to procedures of type `T` must include a value for parameter `i`. If $\mathtt{type}_i$ is omitted, it is taken to be the type of $\mathtt{default}_i$. If both are present, the value of $\mathtt{default}_i$ must be a member of $\mathtt{type}_i$.
- `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 a set of exceptions, the raises set. If "`RAISES {S}`" is omitted, `S` defaults to the set of all exceptions. "`RAISES {}`" means that `S` is the empty set.

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 $\mathtt{signature}_1$ covers $\mathtt{signature}_2$ 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 $\mathtt{signature}_1$ contains the raises set of $\mathtt{signature}_2$.

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

```
VAR p: PROCEDURE(n: INTEGER)
VAR q: PROCEDURE(m: INTEGER)
```

The variables `p` and `q` range over the same set of values, namely the proper procedures with one `VALUE` parameter of type `INTEGER`. But calls that use keyword parameters are interpreted differently; for example, `p(n := 0)` is a valid call, but `p(m := 0)` is not. This interpretation is unchanged by the assignment `p := q`, since assignment changes a variable's value but not its type.

When a series of parameters share the same mode, type, and default, $\mathtt{name}_i$ 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:

$$\mathtt{mode\ v_1,\ ...,\ v_n:\ type\ :=\ default}$$

is shorthand for:

```
mode v₁: type := default; ...; mode vₙ: type := default
```

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

Examples of procedure types:

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

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

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

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 each accept the object as a first argument.

The object type determines the types of a prefix of the fields of the data record, as if "`OBJECT`" were "`REF RECORD`". But the data record can contain additional fields not mentioned in the object type. Similarly, the object type determines the signatures of a prefix of the method suite, but the suite can contain additional methods.

The only way to call a procedure in a method suite is to pass the object itself as the first argument. Consequently, the first parameter to the procedure can be of any type that contains the object. The rest of the procedure signature must be covered by the method declaration in the object type. More precisely, a procedure `p` *satisfies* a method declaration with signature `sig` for an object `x` if `p` is `NIL` or if:

- `p` is a top-level procedure whose first parameter has mode `VALUE` and a type that contains `x`, and
- if `p`'s first parameter is dropped, the resulting signature is covered by `sig`.

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). Such invocations are the only way to access methods.

There are two built-in 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 FieldList METHODS MethodList END
```

where `ST` is an optional supertype, `FieldList` is a list of field declarations, exactly as in a record type, and `MethodList` is a list of *method declarations* and *method overrides*, which are defined below. The names introduced in `FieldList` and `MethodList` must be distinct. If `ST` is omitted, it defaults to `ROOT`. If `ST` is untraced, then the fields must not include traced types.[2]

In both the traced and untraced cases, 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.

The "`:= proc`" is optional. If present, it specifies a default method value used when allocating objects of type `T`; if absent, the default method value is `NIL`. A procedure is a legal default value for method `m` in type `T` if it satisfies the method signature for any object of type `T`; that is, if its first parameter has mode `VALUE` and type some supertype of `T` and if dropping its first parameter results 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 and `proc` is a top-level procedure constant that is a legal default for method `m` in type `T`. Each method override specifies that `proc` is the default value used for method `m` when allocating objects of type `T`. If a method is not overridden, its default in `T` is the same as its default in the supertype.

An object `x` is a member of the type `T` if its data record contains the fields of the supertype, followed by the fields declared in `FieldList`, possibly followed by other fields; its method suite contains procedures that satisfy the method declarations in the supertype, followed by procedures that satisfy the method declarations in `MethodList`, possibly followed by other procedures; its reference class is the same as the reference class of the supertype; and the allocated type of `x` is a subtype of `T`. All fields and methods must appear in the declared order.

Note that the method signatures are statically determined by an object's type (except

[2] This restriction is lifted in unsafe modules.

for the first argument), but the method values are not determined until the object is allocated. They cannot be changed thereafter.

If `T` is an object type and `m` is the name of one of `T`'s methods, then `T.m` denotes `T`'s default `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.

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) = ... ;
```

Since neither `A` nor `AB` has a default value for the `p` method, the method value should be specified when the objects are allocated. The procedures `Pa` and `Pab` are suitable values for the `p` method of objects of types `A` and `AB`. For example:

```
NEW(AB, p := Pab)
```

allocates an object with an `AB` data record and a method that expects an `AB`; it is an example of an object of type `AB`. Similarly,

```
NEW(A, p := Pa)
```

allocates an object with an `A` data record and a method that expects an `A`; it is an example of an object of type `A`. A more interesting example is:

```
NEW(AB, p := Pa)
```

which allocates an object with an `AB` data record and a method that expects an `A`. Since every `AB` is an `A`, the method is not too choosy for the object in which it is placed. The result is a valid object of type `AB`. In contrast,

```
NEW(A, p := Pab)
```

attempts to allocate an object 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 object in which it is placed. The result would not be a member of the type `AB`, so this call to `NEW` is a static error.

Here is an example of default method values and method overrides:

```
TYPE Window =
  OBJECT
    extent: Rectangle
  METHODS
    mouse(e: ClickEvent) := IgnoreClick;
    repaint(e: RepaintEvent) := IgnoreRepaint
  END;

TYPE TextWindow =
  Window OBJECT
    text: TEXT;
    style: TextWindowStyle
  METHODS
    repaint := RepaintTextWindow
  END;
```

`TextWindow` overrides the repaint method but not the mouse method. So if no methods are specified when an object of type `TextWindow` is allocated, its mouse method will be `IgnoreClick` and its repaint method will be `RepaintTextWindow`. The procedure `RepaintTextWindow` can demand a `TextWindow` as its first parameter, but `IgnoreRepaint` and `IgnoreClick` must accept any `Window`.

Finally, an example that uses objects for 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, 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`. Notice that the runtime check on the result of `Delete(q)` is not redundant, since other subtypes of `QueueElem` can be inserted into `q`.

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.

`[u..v] <: B`	if `u` and `v` have basetype `B`
`[u..v] <: [u'..v']`	if `[u..v]` is a (possibly empty) subset of `[u'..v']`

That is, subtyping on ordinal types reflects the subset relation on the value sets.

$$\begin{array}{l} \quad \texttt{(ARRAY OF)}^{m}\ \texttt{ARRAY}\ \texttt{J}_1,\ \ldots,\ \texttt{J}_n\ \texttt{OF ARRAY}\ \texttt{K}_1,\ \ldots,\ \texttt{K}_p\ \texttt{OF T} \\ \texttt{<: (ARRAY OF)}^{m+n}\ \texttt{ARRAY}\ \texttt{I}_1,\ \ldots,\ \texttt{I}_p\ \texttt{OF T} \\ \text{if } \texttt{NUMBER}(\texttt{I}_i) = \texttt{NUMBER}(\texttt{K}_i) \text{ for } i = 1, \ldots, p. \end{array}$$

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, or `A` is fixed and `A'` is open, or they are both fixed and have the same size.

```
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.

`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 `(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 subtype (and therefore of every object type), and every subtype is included in its supertype.

`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; so the types `AT` and `AU` are unrelated by the subtype relation.

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, page 31. Their properties are specified in the required interfaces `Text` and `Thread` (Section 8).

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 Hodges

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 `RETURN` and `EXIT` 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 ten thousand instructions per exception raised to save one instruction per procedure call would be defensible.

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*)`.

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`[3], 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`.

[3] This restriction is lifted in unsafe modules.

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 a variable of type `T`. (If `T` is not an open array type, it follows 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, page 39), 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:

```
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 *)
```

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 into a keyword binding 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 actuals, where `name` is the name of the parameter and `default` is its default value.

The rewritten list of actuals must bind only formal parameters and must bind each formal parameter exactly once.

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.

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 listed in its `RAISES` clause[4] 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:

---

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

```
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)
```

For examples of procedure calls, 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(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.

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)
```

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, page 28.)

Sequential composition

A statement of the form:

```
S1; S2
```

executes S_1, and then if the outcome is normal, executes S_2. If the outcome of S_1 is an exception, S_2 is ignored.[5]

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 of the statement is the exception `e` paired with the argument `x`.

TRY EXCEPT

A `TRY-EXCEPT` statement has the form:

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

where `Body` and each `Handler` are statements, each `id` names an exception, and each `v` is an identifier. The "`ELSE` $\texttt{Handler}_0$" and each "$(\texttt{v}_i)$" 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 $\texttt{id}_i$, then $\texttt{Handler}_i$ is executed. If `Body` raises any other exception and "`ELSE` $\texttt{Handler}_0$" is present, then it is executed. In either case,

[5] Some programmers use the semicolon as a statement terminator, some use it as a statement separator. Similarly, some use the vertical bar in case statements as a prefix operator, some use it as a separator. Modula-3 allows both styles. This report uses both operators as separators.

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:

    id₁, ..., idₙ => Handler

is shorthand for:

    id₁ => Handler; ...; idₙ => 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.

TRY FINALLY

A statement of the form:

    TRY S₁ FINALLY S₂ 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`$_2$.

LOOP

A statement of the form:

    LOOP S END

repeatedly executes `S` until it raises the exit-exception. The statement is equivalent to:

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

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
```

RETURN

A **RETURN** statement for a proper procedure has the form:

```
RETURN
```

The statement raises the return-exception without an argument. It is allowed only in the body of a proper procedure.

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
```

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, it 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).

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
```

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
```

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 equivalent to a procedure call of the form `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`. Free variables in `S` are interpreted in the context of the `WITH` statement. Any `RETURN` or `EXIT` statements in `S` are also interpreted in the context of the `WITH`.

A single `WITH` can contain multiple bindings, which are evaluated sequentially. Thus:

$$\texttt{WITH id}_1\texttt{ = e}_1\texttt{, id}_2\texttt{ = e}_2\texttt{, ...}$$

is equivalent to:

$$\texttt{WITH id}_1\texttt{ = e}_1\texttt{ DO WITH id}_2\texttt{ = e}_2\texttt{ DO ...}$$

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
```

CASE

A **CASE** statement has the form:

```
CASE Expr OF
  L_1 => S_1
| ...
| L_n => S_n
ELSE S_0
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.

TYPECASE

A **TYPECASE** statement has the form:

```
TYPECASE Expr OF
  T_1 (v_1) => S_1
| ...
| T_n (v_n) => S_n
ELSE S_0
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)**" 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 S_0 is present, then it is executed. If the value is a member of no listed type and 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:

```
T_1, ..., T_n => S
```

is shorthand for:

```
T_1 => S | ... | T_n => 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;
```

LOCK

A `LOCK` statement has the form:

```
LOCK mu DO S END
```

where `S` is a statement and `mu` is an expression whose type is `MUTEX` (Section 2, page 16). 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`.

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[6] 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
```

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

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, page 32). Except for variable initializations, the order of declarations in a block does not matter.

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`.

Constants

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

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

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 present, `T` must contain `C`.

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 v₁, ..., vₙ: T := E
```

is shorthand for:

```
VAR v₁: T := E; ...; VAR vₙ: T := E
```

This means that `E` is evaluated `n` times.

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 with signature `sig`, body `B`, and environment 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 initial 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.

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. Exception declarations are allowed only at the top level of interfaces and modules. All declared exceptions are distinct.

Opaque types

An opaque type declaration has the form:

```
TYPE T <: U
```

where `T` is an identifier and `U` an expression denoting a reference type. It declares `T` as a name for some unspecified subtype of `U`.

Revelations

A *revelation* provides information about an opaque type. Unlike other declarations, revelations introduce no new names.

There are two kinds of revelations, *definitive* and *partial*. There can be any number of partial revelations for an opaque type; there must be exactly one definitive revelation.

A definitive 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, `T` is an identifier (possibly qualified by a module name) that has been declared as an opaque subtype of some type `U`, and `V <: U`. Within the scope of the revelation, `T` is known to be the concrete type `V`. The requirement that `V` be branded guarantees that all opaque types in a program are distinct.

A partial revelation has the form:

```
REVEAL T <: V
```

where `V` is a type expression, `T` is a (possibly qualified) identifier that has been declared as an opaque subtype of some type `U`, and `V <: U`. Within the scope of this revelation, `T` is known to be a subtype of `V`. It is a static error if this is not the case.

Revelations are allowed only at the top level of interfaces and modules. A revelation in an interface can be imported into any scope where it is required.

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 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;
VAR v := BITSIZE(v)
```

Examples of illegal declarations:

```
TYPE
  T = RECORD u: U END; U = RECORD t: T END;
  X = X OBJECT END;
CONST
  N = N+1;
TYPE
  T <: ROOT; U <: ROOT;
REVEAL T = U OBJECT END; U = T OBJECT END;
VAR v := P();
PROCEDURE P(): ARRAY [0..NUMBER(v)] OF INTEGER
```

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. A name is visible in a block only if it is declared in the block or in some enclosing block; a name is visible in a module only 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 names in `I` visible in `X`.

Import statements

There are two forms of `IMPORT` statement:

$\texttt{IMPORT I}_1\texttt{, ..., I}_n$

$\texttt{FROM I IMPORT N}_1\texttt{, ..., N}_m$

where the `I`'s are names of interfaces and the `N`'s are names of entities declared in the interface `I`.

The first form makes the names of the interfaces $\texttt{I}_1\texttt{, ..., I}_n$ visible. To refer to an entity named `N` in any $\texttt{I}_j$ the importer must use the qualified identifier $\texttt{I}_j\texttt{.N}$.

The second form makes the names $\texttt{N}_1\texttt{, ..., N}_m$ from interface `I` visible. It does not make the name `I` visible; the importer must refer to $\texttt{N}_i$ and not $\texttt{I.N}_i$.

The same interface can be imported using both forms, in which case both the interface name and the explicitly imported names are visible. Importing an interface provides access to the names and revelations it declares, but not to those it imports.

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.

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, page 9.) 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;          INTERFACE J;          MODULE M EXPORTS I, J;
  PROCEDURE X();        PROCEDURE X();        PROCEDURE X();
```

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

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

But the following is legal, although peculiar:

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

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`.)

Initialization

A *program* is a collection of modules in which no procedure 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 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` *uses* an interface `I` if `M` imports or exports `I` or if `M` uses an interface that imports `I`.
- 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`.

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

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 import dependencies do 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.

Safety

The keyword **`UNSAFE`** can precede the declaration of any interface or module to indicate that it is *unsafe*; that is, that it uses unsafe features of the language (Section 7, page 48). 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.

Example module and interface

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

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

MODULE Stack;
  REVEAL T = BRANDED OBJECT item: REAL; link: T END;
  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;
  PROCEDURE Create(): T = BEGIN RETURN NIL END Create;
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, and Create as before *)
BEGIN
END Stack.
```

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 can be 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`.

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
```

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: REAL)     : REAL
    (x: LONGREAL) : LONGREAL
```

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.

Operation syntax

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

`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:

`M.F(x)`	`(M.F)(x)`	dot before application
`Q(x)^`	`(Q(x))^`	application 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.

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`.[7] This section defines these operations and specifies the conditions under which they produce designators.

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

`r^` denotes the 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 an open reference type, object type, or opaque type, and a checked runtime error if `r` is `NIL`. The type of `r^` is the referent type of `r`.

`a[i]` denotes element `i - FIRST(a)` 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 `T`'s default `m` method. 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 not 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 multi-dimensional 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)`.

If the element type of `a` is packed, implementations may restrict or prohibit the `SUBARRAY` operation.

Numeric literals

Numeric literals denote constant non-negative integers or reals. The types of these literals are `INTEGER`, `REAL`, and `LONGREAL`.

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.

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 preceded by a `+` or `-`. The literal denotes `decimal` times ten raised to the given `exponent`. If "`E exponent`" is omitted, `exponent` defaults to `0`.

A literal `LONGREAL` has the form `decimal D exponent`, and `decimal` and `exponent` are the same as in a literal `REAL`.

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`.

Text and character literals

A character literal is a single printing character or escape sequence enclosed in single quotes. The type of a character literal is `CHAR`.

A text literal is a sequence of zero or more printing characters or escape sequences enclosed in double quotes. The type of a text literal is `TEXT`.

Printing characters are all printing ISO-Latin-1 characters except double-quote, right single quote, and backslash (see Section 9, page 58).

The only way to include a quotation mark or non-printing character such as newline in a text or character literal is with an escape sequence. Here are the legal escape sequences:

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

NIL

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

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.

Set, array, and record constructors

A set constructor has the form:

$\texttt{S\{e}_1\texttt{, ..., e}_n\texttt{\}}$

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:

$\texttt{A\{e}_1\texttt{, ..., e}_n\texttt{\}}$

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 multi-dimensional array, the `e`'s must themselves be array-valued expressions.

If `A` is a fixed array type and `n` is at least 1, then $\texttt{e}_n$ can be followed by "`..`" to indicate that the value of $\texttt{e}_n$ 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). 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.

NEW

An allocation operation has the form:

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

where `T` is a reference type other than `REFANY`, `ADDRESS`, or `NULL`. If `T` is a fixed reference type, the operation returns the address of a newly-allocated variable of `T`'s referent type. If `T` has an object type, the operation returns a newly-allocated data record containing the data fields declared in `T`, paired with a method suite containing the methods declared in `T`. 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 the scope of a definitive revelation for `T` (Section 4, page 31).

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:

$$\texttt{NEW(T, } \texttt{n}_1\texttt{, ..., } \texttt{n}_k\texttt{)}$$

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 type.

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 and methods. Positional bindings are not allowed.

Each binding `f := v` initializes the field or method `f` to the value `v`. Fields or methods 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. Note that methods always have defaults, since the "default default" is `NIL`. It is a static error if any method's signature fails to satisfy its declaration (Section 2, page 11), and a checked runtime error if any method is not a top-level procedure.

Arithmetic operations

The effect of an operation that overflows, underflows, or divides by zero is a checked runtime error. To perform arithmetic operations modulo the word size, programs should use the routines in the `Word` interface (page 52).

prefix `+ (x: INTEGER) : INTEGER`

```
            (x: REAL)         : REAL
            (x: LONGREAL)     : LONGREAL
```

```
infix   +   (x,y: INTEGER)    : INTEGER
            (x,y: REAL)       : REAL
            (x,y: LONGREAL)   : LONGREAL
            (s,t: 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, `x IN (s + t)` if and only if `(x IN s) OR (x IN t)`. The types of `s` and `t` 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: REAL)         : REAL
            (x: LONGREAL)     : LONGREAL
```

```
infix   -   (x,y: INTEGER)    : INTEGER
            (x,y: REAL)       : REAL
            (x,y: LONGREAL)   : LONGREAL
            (s,t: 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: `x IN (s - t)` if and only if `(x IN s) AND NOT (x IN t)`. The types of `s` and `t` 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: REAL)       : REAL
            (x,y: LONGREAL)   : LONGREAL
            (s,t: Set)        : Set
```

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

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

On reals, `/` denotes division. On sets, it denotes symmetric difference. In other words, `x IN (s / t)` if and only if `(x IN s) # (x IN t)`. The types of `s` and `t` 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
```

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`. 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`.

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

`ABS(x)` is the absolute value of `x`.

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

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

`FLOAT(x)` is the nearest `REAL` to `x`; `LONGFLOAT(x)` is the nearest `LONGREAL`. Ties are broken arbitrarily.

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

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

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

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

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

`ROUND(r)` is the nearest integer to `r`; ties are broken arbitrarily. `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: REAL)      : REAL
          (x,y: LONGREAL)  : LONGREAL
```

`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.

Relations

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

The operator `=` returns `TRUE` if `x` and `y` have the same value. The operator `#` returns `TRUE` if `x` and `y` have different values. It is a static error if the type of `x` is not assignable to the type of `y` or vice versa.

References are equal if they address the same variable. 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: REAL)      : BOOLEAN
                (x,y: LONGREAL)  : BOOLEAN
                (x,y: ADDRESS)   : BOOLEAN
                (x,y: Set)       : BOOLEAN
```

In the first four cases, `<=` returns `TRUE` if `x` is not greater than `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: REAL)      : BOOLEAN
              (x,y: LONGREAL)  : BOOLEAN
              (x,y: ADDRESS)   : BOOLEAN
              (x,y: Set)       : BOOLEAN
```

In all cases, `x < y` means `(x <= y) AND (x # y)`, and `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.

```
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`.

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.

Type operations

```
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))`. Similarly, 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)
       (A: FixedArrayType)  : BaseType(IndexType(A))
       (a: Array)           : BaseType(IndexType(a))

LAST   (T: OrdinalType)     : BaseType(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 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`.

```
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)        : INTEGER
          (r: REFANY)         : INTEGER
          (r: UNTRACED ROOT)  : INTEGER
```

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.

```
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.

Text operations

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

The concatenation of `a` and `b`, as defined by `Text.Cat`. (Section 8, page 50.)

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 function application. All required operations in the `Word` interface (Section 8, page 52) are allowed in constant expressions.

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. All literals are legal in constant expressions; procedure constants are not.

7 Unsafe operations

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

The features defined in this chapter 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 with a fixed or object reference type. 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 Required interfaces

C++ has a host of operators that will be explained if and where needed.
—The C++ Programming Language

Modula-3 requires that every implementation provide the interfaces `Text`, `Thread`, and `Word`, as specified in this chapter. Implementations are free to extend these interfaces, as long as they do not invalidate clients of the unextended interfaces.

The Text interface

For a semantic specification of an extended version of the **Text** interface, see "The Modula-2+ Text Interface" [2].

```
INTERFACE Text;

TYPE
  T = TEXT;
  (* A zero-based sequence of characters. It is a checked runtime
     error to pass the NIL TEXT to any procedure in this interface. *)

PROCEDURE Cat(t, u: T): T;
(* The concatenation of t and u.  *)

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;
(* The number of characters in t. *)

PROCEDURE Empty(t: T): BOOLEAN;
(* TRUE if Length(t) = 0. *)

PROCEDURE Sub(t: T; start, length: CARDINAL): T;
(* Return a subsequence of t: empty if start >= Length(t)
   or length = 0; otherwise the subsequence ranging from
   start to MIN(start+length, Length(t) - 1). *)

PROCEDURE SetChars(VAR a: ARRAY OF CHAR; t: T);
(* For each i [0..MIN(LAST(a), Length(t) - 1)], set a[i] to
   GetChar(t, i). *)

PROCEDURE FromChar(ch: CHAR): T;
(* A text containing the single character ch. *)

PROCEDURE FromChars(READONLY a: ARRAY OF CHAR): T;
(* A text containing the characters of a. *)

END Text.
```

The Thread interface

If a shared variable is written concurrently by two threads, or written by one and read concurrently by another, the effect is to set the variable to an implementation-dependent value of its type. For example, if one thread writes `a[0]` while another concurrently writes `a[1]`, one of the writes might be lost. Thus, portable programs must use the `Thread` interface to provide mutual exclusion for shared variables. The formal specification of this interface is in "Synchronization Primitives for a Multiprocessor: A Formal Specification" [1].

```
INTERFACE Thread;

TYPE
  T         <: REFANY;  (* A handle on a thread *)
  Mutex     =  MUTEX;   (* Locked by some thread, or unlocked *)
  Condition <: ROOT;    (* A set of waiting threads *)

  (* Initially a Mutex is unlocked and a Condition is empty.
     It is a checked runtime error to pass the NIL Mutex,
     Condition, or T to any procedure in this interface. *)

  Closure = OBJECT METHODS apply(): REFANY RAISES {} END;

PROCEDURE NewMutex(): Mutex;
(* A newly-allocated, unlocked mutex *)

PROCEDURE NewCondition(): Condition;
(* A newly-allocated condition with no threads waiting on it. *)

PROCEDURE Fork(cl: Closure): T;
(* A handle on a newly-created thread executing cl.apply(). *)

PROCEDURE Join(t: T): REFANY;
(* Wait until t has terminated and return its result.  It is a
   checked error to call this more than once for any t. *)

PROCEDURE Wait(m: Mutex; c: Condition);
(* The 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);
(* The calling thread must have m locked.  Unlocks m. *)

PROCEDURE Broadcast(c: Condition);
```

```
(* All threads waiting on c become eligible to run.  *)

PROCEDURE Signal(c: Condition);
(* One or more threads waiting on c become eligible to run. *)

PROCEDURE Self(): T;
(* Return the handle of the calling thread. *)

EXCEPTION Alerted; (* Approximate asynchronous interrupts *)

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
   call or sometime during the wait, lock m and raise Alerted.
   Implementations may raise additional exceptions. *)

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, raise Alerted.
   Implementations may raise additional exceptions. *)

CONST
  AtomicSize = ...;
(* An implementation-dependent integer constant:  the number of bits
   in a memory-coherent block.  If two components of a record or
   array fall in different blocks, they can be accessed concurrently
   by different threads without locking. *)

END Thread.
```

The Word interface

```
INTERFACE Word;

(* A Word.T w represents a sequence of Word.Size bits
```

$$w_0, \ldots, w_{\mathtt{Word.Size-1}}.$$

```
It also represents the unsigned number
```

$$\text{sum of } 2^i * w_i \text{ for } i \text{ in } 0, \ldots, \mathtt{Word.Size-1}. \ *)$$

```
TYPE T = INTEGER;  (* encoding is implementation-dependent;
                      for example, two's complement. *)

CONST Size = BITSIZE(T);

PROCEDURE Plus (x,y: T): T;     (* (x + y) MOD 2^Word.Size *)

PROCEDURE Times (x,y: T): T;    (* (x * y) MOD 2^Word.Size *)

PROCEDURE Minus (x,y: T): T;    (* (x - y) MOD 2^Word.Size *)

PROCEDURE Divide(x,y: T): T;    (* x divided by y *)

PROCEDURE Mod (x,y: T): T;      (* x MOD y *)

PROCEDURE LT(x,y: T): BOOLEAN;  (* x < y *)

PROCEDURE LE(x,y: T): BOOLEAN;  (* x <= y *)

PROCEDURE GT(x,y: T): BOOLEAN;  (* x > y *)

PROCEDURE GE(x,y: T): BOOLEAN;  (* x >= y *)

PROCEDURE And(x,y: T): T;       (* Bitwise AND of x and y *)

PROCEDURE Or (x,y: T): T;       (* Bitwise OR of x and y *)

PROCEDURE Xor(x,y: T): T;       (* Bitwise XOR of x and y *)

PROCEDURE Not (x: T): T;        (* Bitwise complement of x *)


PROCEDURE Shift(x: T; n: INTEGER): T;
(* For all i such that both i and i - n are in the range
   [0..Word.Size - 1], bit i of the result equals bit i - n of x.
   The other bits of the result are 0.  Thus shifting by n > 0 is
   like multiplying by 2^n *)

PROCEDURE Rotate(x: T; n: INTEGER): T;
(* Bit i of the result is bit ((i - n) MOD Word.Size) of x. *)

PROCEDURE Extract(x: T; i, n: CARDINAL): T;
(* Take n bits from x, with bit i as the least significant
   bit, and return them as the least significant n bits of
   a word whose other bits are 0.  A checked runtime error
   if n + i > Word.Size. *)
```

```
PROCEDURE Insert(x: T; y: T; i, n: CARDINAL): T;
(* Result of replacing n bits of x, with bit i as the least
   significant  bit, by the least significant n bits of y.
   The other bits of x are unchanged.  A checked runtime
   error if n + i > Word.Size. *)

END Word.
```

9 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 Hodges

Keywords

Here are the Modula-3 keywords:

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

Reserved identifiers

Here are the *reserved* identifiers, which cannot be redeclared:

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

Operators

The following characters and character pairs are classified as operators:

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

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.

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 L*>` precedes a procedure declaration to indicate that the procedure is implemented in the language `L`, or precedes an interface to indicate that the entire interface is implemented in the language `L`.

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`

Parentheses are used for grouping. Non-terminals begin with an upper-case letter. Terminals are either keywords or quoted operators. The symbols `Ident`, `Number`, `TextLiteral`, and `CharLiteral` are defined in the token grammar on page 58. Each production is terminated by a period. Indented productions are used only in the productions immediately above them. The syntax does not reflect the restrictions that revelations and exceptions can only be declared at the top level; nor does it include explicit productions for `NEW`, `INC`, and `DEC`, which parse like procedure calls.

Compilation unit productions

```
Compilation = [ UNSAFE ] ( Interface | Module ).
Interface   = INTERFACE Ident ";" { Import }
                 { Declaration } END Ident ".".
Module      = MODULE Ident [ EXPORTS IDList ] ";" { Import }
                 Block Ident ".".
Import      = [ FROM Ident ] IMPORT IDList ";".
Block       = { Declaration } BEGIN Stmts END.
Declaration = CONST { ConstDecl ";" }
            | TYPE { TypeDecl ";" }
            | EXCEPTION { ExceptionDecl ";" }
            | VAR { VariableDecl ";" }
            | ProcedureHead [ "=" Block Ident ] ";"
            | REVEAL { TypeID ( "=" | "<:" ) Type ";" }.

ConstDecl     = Ident [ ":" Type ] "=" ConstExpr.
TypeDecl      = Ident ( "=" | "<:" ) Type.
ExceptionDecl = Ident [ "(" Type ")" ].
VariableDecl  = IDList (":" Type & ":=" Expr).
ProcedureHead = PROCEDURE Ident Signature.
Signature     = "(" Formals ")" [ ":" Type ] [ RAISES Raises ].

Formals = [ Formal { ";" Formal } [ ";" ] ].
Formal = [ VALUE | VAR | READONLY ] IDList ( ":" Type & ":=" ConstExpr ).
Raises = "{" [ ExceptionID { "," ExceptionID } ] "}".
```

Statement productions

```
Stmts =  [ Stmt { ";" Stmt } [ ";" ] ].

Stmt =  AssignStmt | Block | CallStmt | CaseStmt
     | ExitStmt | EvalStmt | ForStmt | IfStmt | LockStmt
     | LoopStmt | RaiseStmt | RepeatStmt | ReturnStmt
     | TryFinStmt | TryXptStmt | TCaseStmt | WhileStmt | WithStmt.

AssignStmt = Expr ":=" Expr.
CallStmt   = Expr "(" [ Actual { "," Actual } ] ")".
CaseStmt   = CASE Expr OF [ Case ] { "|" Case } [ELSE Stmts] END.
ExitStmt   = EXIT.
EvalStmt   = EVAL Expr.
ForStmt    = FOR Ident ":=" Expr TO Expr [ BY Expr ] DO Stmts END.
IfStmt     = IF Expr THEN Stmts {ELSIF Expr THEN Stmts} [ELSE Stmts] END.
LockStmt   = LOCK Expr DO Stmts END.
LoopStmt   = LOOP Stmts END.
```

```
RaiseStmt  = RAISE ExceptionID [ "(" Expr ")" ].
RepeatStmt = REPEAT Stmts UNTIL Expr.
ReturnStmt = RETURN [ Expr ].
TCaseStmt  = TYPECASE Expr OF [ Tcase ] { "|" Tcase } [ELSE Stmts] END.
TryXptStmt = TRY Stmts EXCEPT [Handler] {"|" Handler} [ELSE Stmts] END.
TryFinStmt = TRY Stmts FINALLY Stmts END.
WhileStmt  = WHILE Expr DO Stmts END.
WithStmt   = WITH Binding { "," Binding } DO Stmts END.

Case    = Labels { "," Labels } "=>" Stmts.
Labels  = ConstExpr [".." ConstExpr].
Handler = ExceptionID { "," ExceptionID } ["(" Ident ")"] "=>" Stmts.
Tcase   = Type { "," Type } [ "(" Ident ")" ] "=>" Stmts.
Binding = Ident "=" Expr.

Actual  = ( [ Ident ":=" ] Expr | Type ).
```

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    = [Ancestor] [Brand] OBJECT Fields [ METHODS Methods ] END.
ProcedureType = PROCEDURE Signature.
RecordType    = RECORD Fields END.
RefType       = [ UNTRACED ] [ Brand ] REF Type.
SetType       = SET OF Type.
SubrangeType  = "[" ConstExpr ".." ConstExpr "]".

Ancestor = TypeName | ObjectType | UNTRACED.
Brand    = BRANDED [ TextLiteral ].
Fields   = [ Field { ";" Field } [ ";" ] ].
Field    = IDList ( ":" Type & ":=" ConstExpr ).
Methods  = [ Method { ";" Method } [ ";" ] ].
Method   = Ident ( Signature & ":=" ProcedureID ).
```

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 = Ident | Number | CharLiteral | TextLiteral
    | Constructor | "(" Expr ")".

Relop = "=" | "#" | "<" | "<=" | ">" | ">=" | IN.
Addop = "+" | "-" | "&".
Mulop = "*" | "/" | DIV | MOD.

Selector = "^" | "." Ident | "[" Expr { "," Expr } "]"
    | "(" [ Actual { "," Actual } ] ")".

  Constructor = Type "{" [ SetCons | RecordCons | ArrayCons ] "}".

SetCons = SetElt { "," SetElt }.
SetElt = Expr [ ".." Expr ].
RecordCons = RecordElt { "," RecordElt }.
RecordElt = [ Ident ":=" ] Expr.
ArrayCons = Expr { "," Expr } [ "," ".." ] .
```

Miscellaneous productions

```
TypeName = Ident [ "." Ident ] | ROOT | UNTRACED ROOT.
ExceptionID = Ident [ "." Ident ].
ProcedureID = Ident [ "." Ident ].

IDList = Ident { "," Ident }.
```

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 9, page 55) or an **Ident** or **Literal**, as defined here.

An **Ident** is a case-significant sequence of letters, digits, and underscores that begins with a letter. An **Ident** is a keyword if it appears in Section 9, a reserved identifier if it appears in Section 9, 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.

```
Literal = Number | CharLiteral | TextLiteral.

Ident = Letter { Letter | Digit | "_" }.

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

CharLiteral = "'"  ( PrintingChar | Escape ) "'".

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 } [ Exponent ].

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

PrintingChar = Letter | Digit | OtherChar.

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

Digit = "0" | "1" | ... | "9".

OctalDigit = "0" | "1" | ... | "7".

Letter = "A"  | "B"  | ... | "Z"  | "a"  | "b"  | ... | "z".

OtherChar = " " | "!" | "#" | "$" | "%" | "&" | "(" | ")"
          | "*" | "+" | "," | "-" | "." | "/" | ":" | ";"
          | "<" | "=" | ">" | "?" | "@" | "[" | "]" | "^"
          | "_" | "`" | "{" | "|" | "}" | "~"
          | ExtendedChar

ExtendedChar = any char with ISO-Latin-1 code in [8_240..8_377].
```

Acknowledgments

Modula-3 was designed by Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson, as a joint project by the Digital Systems Research Center and the Olivetti Research Center.

Paul Rovner made many contributions as a founding member of the design committee, but cannot be held responsible for the final product.

Our starting point was Modula-2+, which was designed by Paul Rovner, Roy Levin, John Wick, Andrew Birrell, Butler Lampson, and Garret Swart. We benefited from the ruthlessly complete description of Modula-2+ provided in Mary-Claire van Leunen's *Modula-2+ User's Manual*.

Niklaus Wirth made valuable suggestions and inspired us with the courage to throw things out. He also designed Modula-2, the starting point of our starting point.

Too many people provided helpful feedback to thank individually, but we especially want to thank the following people: Bob Ayers, Andrew Black, David Chase, Dan Craft, Hans Eberle, John Ellis, Stu Feldman, Jim Horning, Mike Kupfer, Butler Lampson, Trevor Morris, Eric Muller, Lyle Ramshaw, Eric Roberts, Ed Satterthwaite, Jorge Stolfi, and Garret Swart.

The report was written by Lucille Glassman and Greg Nelson, under the watchful supervision of the whole committee.

Bibliography

[1] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. *Operating Systems Review*, 21(5), November 1987. Also published as SRC Research Report 20, August 1987.

[2] Daniel Jackson and Jim Horning. The Modula-2+ Text interface. Unpublished manuscript available from Jim Horning at SRC.

[3] Butler W. Lampson. A description of the cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, December 1983.

[4] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language euclid. Technical Report CSL-81-12, Xerox Palo Alto Research Center, October 1981.

[5] Larry Tesler, Apple Computers. Object pascal report. *Structured Language World*, 9(3), 1985.

[6] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-78-1, Xerox Palo Alto Research Center, February 1978.

[7] Paul Rovner. Extending modula-2 to build large, integrated systems. *IEEE Software*, 3(6), November 1986.

[8] Paul Rovner, Roy Levin, and John Wick. On extending modula-2 for building large, integrated systems. Research Report 3, Digital Systems Research Center, January 1985.

[9] N. Wirth. From modula to oberon and the programming language oberon. Technical Report 82, Institut fur Informatik, ETH Zurich, September 1987.

[10] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Third Edition, 1985.

Index