31

Modula-3 Report

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

August 25, 1988

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

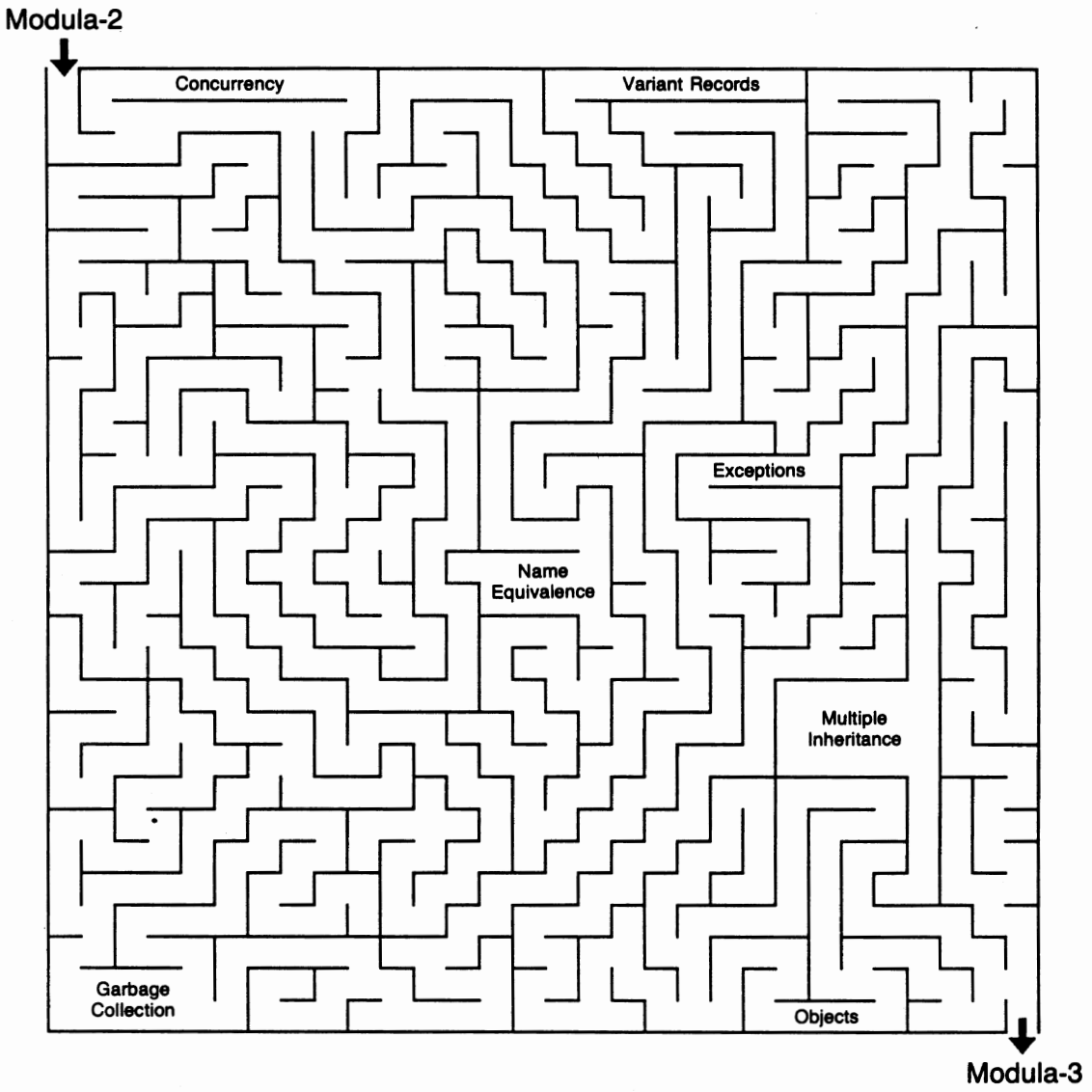
DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Modula-3 Report

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

August 25, 1988



Authors' abstract

This report defines the programming language Modula-3, which was designed by the DEC Systems Research Center and the Olivetti Research Center, with the guidance and inspiration of Niklaus Wirth.

Capsule review

Modula-3 descends from Mesa, Modula-2, Cedar, and Modula-2+, and resembles Object Pascal, Oberon, and Euclid. A notable feature of this language family is the use of modules to delineate the separation between the implementation and the use of interfaces; modules allow separate compilation without sacrificing the strong typechecking of languages like Pascal. Modula-3 supports object-oriented programming, garbage collection, exception handling, lightweight processes, and the isolation of unsafe features. It incorporates a new type system that is simpler, more uniform, and more powerful than those of its ancestors.

It is unusual to find so many good things in such a small language, and unusual for such a brief language manual to be so complete. The designers were determined that the language would be definable in a readable fifty-page manual. They succeeded admirably; both the language and its exposition are exemplary. This report will certainly interest anyone concerned with modular programming languages, type systems, or programming language definition.

Jim Horning

Table of Contents

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.

We thank the following people for their helpful feedback: Bob Ayers, Andrew Black, David Chase, Dan Craft, Hans Eberle, John Ellis, Jim Horning, Mike Kupfer, Butler Lampson, Lyle Ramshaw, Eric Roberts, Ed Satterthwaite, Jorge Stolfi, and Garret Swart.

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

1. Overview

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? It is true, that what is settled by custom, though it be not good, yet at least it is fit. And those things that have long gone together are as it were confederate within themselves: whereas new things piece not so well; but though they help by their utility, yet they trouble by their inconformity. Besides, they are like strangers, more admired and less favored. All this is true, if time stood still; which contrariwise moveth so round, that a forward retention of custom is as turbulent a thing as an innovation; and they that reverence too much old times are but a scorn to the new.

--- Francis Bacon

Modula-3 descends from Mesa [8], Modula-2 [12], Cedar [5], and Modula-2+ [9, 10]. It also resembles its cousins Object Pascal [11], Oberon [13], and Euclid [6]. Since these languages already have more raw material than fits comfortably into a readable fifty-page language definition, which we were determined to produce, we didn't need to be inventive. On the contrary, we had to leave many good ideas out.

One of our main goals was to provide safety from unchecked runtime errors---forbidden operations that invalidate an invariant of the runtime system and lead to an unpredictable computation.

A classic unchecked runtime error is to free a record to which active references remain. To avoid this danger, Modula-3 follows Cedar, Modula-2+, and Oberon by automatically freeing unreachable records. To allow both garbage collection and low-level systems programming, Modula-3 provides both traced and untraced references.

Another well-known unchecked runtime error is to assign to the tag of a variant record in a way that subverts the type system. Distinguishing subversive assignments from benign assignments in the language definition is error-prone and arbitrary. The objects and classes first introduced by Simula [2] and adopted in Oberon and Object Pascal are more general than variant records, and they are safe, so we have discarded variant records and adopted objects. In Modula-3, all objects are references.

Generally the lowest levels of a system cannot be programmed with complete safety. Neither the compiler nor the runtime system can check the validity of a bus address for an IO controller, nor can they limit the ensuing havoc if it is invalid. This presents the language designer with a dilemma. If he holds out for safety, then low level code will have to be programmed in another language. But if he adopts unsafe features, then his safety guarantee becomes void everywhere. In this area we have followed the lead of Cedar and Modula-2+ by adopting a small number of unsafe features that are allowed only in modules that are explicitly labeled unsafe. In a safe module, the compiler guarantees the absence of unchecked runtime errors; in an unsafe module, it is the programmer's responsibility to avoid them.

From Mesa and Modula-2 we adopted modules, which are separate program units with explicit interfaces, and abstract (or opaque) types, which hide the representation of a type from client modules that use the type. In Modula-3, as in some implementations of Modula-2, variables with opaque types must be references. If the hidden representation changes but the interface remains the same, client modules will not need to be reprogrammed, or even recompiled.

From Modula-2+ we adopted exceptions and threads.

An exception exits all procedure call levels between the point at which it is "raised" and the point at which it is "handled". Exceptions are a good way to handle any runtime error that is not necessarily fatal. The alternative is to use error return codes, but this has the

drawback that programmers don't consistently test for them. In the Unix/C world, the frequency with which programs omit tests for error returns has become something of a standing joke. Instead of breaking at an error, too many programs continue on their unpredictable way. Raising an exception is a better approach, since it will stop the computation unless there is an explicit handler for it.

A thread is a "light-weight process". The threads interface provides a simplified version of the Mesa extensions to Hoare's monitors [1, 3, 7]. Waiting, signaling, and locking a monitor have Hoare's semantics, but the requirement that a monitored data structure be an entire module is relaxed: it can be an individual record or any set of variables instead. The programmer is responsible for acquiring the appropriate lock before accessing the monitored data.

1.1. Common notions

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.

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 `T` is a member of `U`.

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. Every expression has a statically-determined type, which contains every value that can be produced by the expression. In particular, literals have types; for example, the type of `"6"` is `INTEGER` (not `[6..6]`). Expressions whose values can be determined statically are called *constant expressions*.

Expressions that produce variables are called *designators*. The type of a designator is the type of the variable it produces. A designator is *writable* if it can be used in contexts that require mutable variables and *readonly* otherwise.

Every expression has a unique type, but a value can be a member of many types. For example, `6` is a member of both `[0 .. 9]` and `INTEGER`. 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`". But there is one case in which one of the types that contain a value is distinguished: when a traced reference or object is allocated it is tagged with a type, which we call the *allocated type* of the reference value.

The type denoted by a type name never changes at runtime, and is usually determined at compile time. But a type name can be *opaque* in a scope, which means that it denotes a reference type with an unknown referent type. A type name that is not opaque is *concrete*. A concrete type is determined at compile time (except for the referent types of opaque types that occur within it).

Types are distinct until proven identical. For example, variables with types `T` and `U` would not be assignable in a scope where `T` and `U` are both opaque, but would be assignable in any scope where `T` and `U` are identified with the same concrete type.

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 illegal to declare a variable of an empty type.

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.

1.2. Required interfaces

An implementation of Modula-3 must include implementations of the interfaces Text, Thread, Word, and Fmt. The first three provide elementary operations on text strings, concurrent threads of control, and "words", which represent bit vectors or unsigned integers. The Fmt interface provides procedures for formatting numbers and other data as text.

The full name of an entity is the name of its interface followed by a dot and its name in that interface. The principal type in an interface is usually named `T`. Thus `Text.T` is the type for text strings, `Thread.T` is the type for threads, and `Word.T` is the type for words.

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

Two types are identical if their definitions become the same when expanded; that is, when all names in the type definition are replaced by their definitions. In the case of recursive types, the expansion is infinite. In other words, Modula-3 uses structural equivalence, while Modula-2 uses name equivalence. A type expression is allowed wherever a type is required.

2.1. Ordinal types

An *integer* value is a member of an implementation-dependent subrange of the mathematical integers. An *enumeration* value is a member of an ordered collection that is defined by an explicit list of its elements. Integers and enumerations are collectively called *ordinal values.*

Any type whose values are ordinal values is an *ordinal type*. There are two kinds of ordinal types: enumerations and subranges.

The declaration of an enumeration type has the form:

```
TYPE T = {id1, id2, ..., idn}
```

where the `id`'s are distinct identifiers. The type `T` is an ordered set of `n` values; the expression $T.id_i$ denotes the `i`'th value of the type in increasing order. Empty enumerations are allowed.

The declaration of a subrange type has the form:

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

where `Lo` and `Hi` are two integers or two values from the same enumeration. The values of `T` are all the values from `Lo` to `Hi` inclusive. `Lo` and `Hi` must be constant expressions (Section 6.13, page 39). If `Lo` exceeds `Hi`, the subrange is empty.

The operators `ORD` and `VAL` convert between enumerations and integers. The operators `FIRST`, `LAST`, and `NUMBER` can be applied to ordinal types to obtain the first element, last element, and number of elements, respectively (Section 6.11, page 38).

There are four built-in ordinal types:

`INTEGER`	The type containing all integers represented by the implementation
`CARDINAL`	The subrange `[0 .. LAST(INTEGER)]`
`BOOLEAN`	The enumeration `{FALSE, TRUE}`
`CHAR`	An enumeration containing at least 128 elements

The first 128 elements of type `CHAR` are intended to represent characters in the ASCII code. `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, consider the declarations:

```
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 identical types. In particular, `T1.C = T2.C` and therefore `U1` and `U2` are also identical types. 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`.

2.2. Floating-point types

A *floating point* value is a member of an implementation-dependent subset of the mathematical real numbers. There are two built-in floating-point types:

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

2.3. 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 represent sets with bit vectors.

2.4. 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.

Arrays are assignable if they have the same element type and length. If either the source or target is an open array, a runtime length check is required (Section 3.1, page 15).

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`.

Only the number of elements of the index type, not the particular bounds, affects the values of `T`. The bounds are used to determine the interpretation of indexes. 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, and are assignable to one another. 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]`.

An expression of the form:

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

is shorthand for:

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

Similarly, an expression of the form $a[i_1, \ldots, i_n]$ is shorthand for $a[i_1] \ldots [i_n]$.

An open array type declaration has the form:

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

where `Element` is any type. The values of `T` are arrays whose element type is `Element` and whose length is arbitrary. The index set of an open array variable is the integer subrange `[0 .. n-1]`, where `n` is the length of the array. If `Element` is itself an open array type, then all elements of any variable of type `T` will have the same length in all dimensions.

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 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
```

2.5. Records

A *record* `r` is a sequence of named variables, called the *fields* of the record. Different fields can have different types. The sequence of fields of a record is statically determined by its type. The expression `r.f` designates the field `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 type other than an open array type, and `default` is a constant expression. 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 for use in record constructors (page 35). Either `":= default"` or `": Type"` can be omitted, but not both. If `default` is omitted, constructors for records of type `T` must specify a value for `fieldName`. 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 an expansion in which the type and default are repeated for each identifier in the list. That is:

```
f1, ..., fm: Type := default
```

is shorthand for:

```
f1: Type := default; ...; fm: Type := default
```

Examples of record types:

```
TYPE
  Time = RECORD seconds: INTEGER; microseconds: [0 .. 999999] END;
  Alignment = {Left, Center, Right};
  TextWindowStyle = RECORD
    align := Alignment.Center;
    font := Font.Default;
    foreground := Color.Black;
    background := Color.White;
    margin, border := 2
  END
```

2.6. Packed types

A packed type declaration has the form:

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

where `Base` is a type and `n` is an integer-valued constant expression. The values allowed for `n` are implementation-dependent. An illegal value for `n` is a static error.

The values of type `T` are the same as the values of type `Base`, but variables of type `T` that occur in records 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.

2.7. 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 all address variables of some fixed type; the members of an open reference type can 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 2.9, page 10, describes object types.

A reference type is either *traced* or *untraced*. A member of a traced reference type is traced by the garbage collector; that is, the implementation stores its referent in a system-managed storage pool, determines at runtime when all traced references to it are gone, and then reclaims its storage. A member of an untraced reference type is not traced by the garbage collector.

If two reference types are both traced or both untraced, they are of the same *reference class*.

There are three built-in reference types:

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

`REFANY` and `ADDRESS` are the only open reference types. The `TYPECASE` statement

(Section 3.18, page 22) 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. Type 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. Type is called the *referent type* of T.

Examples of reference types:

```
TYPE
  TextLine = REF ARRAY OF CHAR;
  IntList = REF RECORD item: INTEGER; link: IntList END;
  ControllerHandle = UNTRACED REF RECORD
    status: BITS 8 FOR [0 .. 255];
    filler: BITS 12 FOR [0 .. 0];
    pc: BITS 12 FOR [0 .. 4095]
  END
```

2.8. Procedures

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

the *body*, which is a statement,

the *signature*, which specifies the procedure's formal arguments, result type, and raises set (the set of exceptions that the procedure can raise), and

the *environment*, which is the scope with respect to which variable names in the body will be interpreted.

A *top-level* procedure is a procedure declared in the outermost scope of a module. Any other procedure is a *local* procedure. 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 *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:

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

[1]This restriction is lifted in unsafe modules.

where:

Each `mode`$_i$ is a parameter mode, which can be VALUE, VAR, or READONLY. If `mode`$_i$ is omitted, it defaults to VALUE.

Each `name`$_i$ is an identifier, the name of parameter `i`.

Each `type`$_i$ is a type, the type of parameter `i`.

Each `default`$_i$ is a constant expression to be used as a default value for parameter `i`. If `mode`$_i$ is VAR, it is a static error to include `":= default`$_i$". If `mode`$_i$ is READONLY or VALUE, either `":= default`$_i$" or `": type`$_i$" can be omitted, but not both. If `default`$_i$ is omitted, calls to procedures of type `T` must include a value for parameter `i`. If `type`$_i$ is omitted, it is taken to be the type of `default`$_i$. If both are present, the value of `default`$_i$ must be a member of `type`$_i$.

`R` is the result type, which can be any type but an open array type. The `": R"` can be omitted, in which case the signature is for a proper procedure.

`S` is a set of exceptions, the raises set. `"RAISES {S}"` can be omitted, in which case `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 `signature`$_1$ covers `signature`$_2$ if:

They have the same number of parameters, and corresponding parameters have the same type and mode.

They have identical result types, or neither has a result type.

The raises set of `signature`$_1$ contains the raises set of `signature`$_2$.

The parameter names and defaults are used only in the binding of actuals in calls to the procedure; they are irrelevant to the value of the procedure variable, which is a simple closure. 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, and are assignable to one another. But calls that use keyword parameters are interpreted differently; for example, `p(n := 0)` and `q(m := 0)` are valid calls, but `p(m := 0)` and `q(n := 0)` are not. (Section 3.2, page 16.)

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

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

is shorthand for:

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

Examples of procedure types:

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

2.9. Objects

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

The types of the initial fields of the data record are determined by the object type, as if "OBJECT" were "REF RECORD". But the data record can contain additional fields not mentioned in the object type. Similarly, the signatures of the initial methods of the method suite are determined by the object type, but the suite can contain additional methods.

If o is an object, then o.f designates the data field named f in o's data record. If m is one of o's methods, an invocation of the form o.m(...) denotes an execution of o's m method (Section 3.2, page 16). Invocations of this form are the only way in which methods can be accessed.

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 procedure signature is covered by sig.

An object type can be traced or untraced and can be declared with or without a supertype.

2.9.1. The declaration of a traced object type without a supertype has the form:

```
TYPE T = OBJECT FieldList METHODS MethodList END
```

where FieldList is a list of field declarations, exactly as in a record type, and MethodList is a list of method declarations. Each 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.

An object x is a member of the type T if it is NIL or a traced reference to a data record that contains the fields declared in FieldList, in the declared order, possibly followed by other fields, paired with a method suite that contains procedures that satisfy the method declarations in MethodList, in the declared order, possibly followed by other procedures, and the allocated type of x is a subtype of T.

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.

Note that the method signatures are statically determined by an object's type (except for the first argument), but the method values are not determined until the object is allocated. They cannot be changed thereafter.

2.9.2. The declaration of an untraced object type without a supertype has the form:

```
TYPE T = UNTRACED OBJECT FieldList METHODS MethodList END
```

where `FieldList` is a list of field declarations with untraced[2] types and `MethodList` is as above. The only effect of `"UNTRACED"` is that the objects in `T` are untraced references.

2.9.3. The declaration of an object type with a supertype has the form:

```
TYPE T = Supertype OBJECT FieldExtension METHODS MethodRevision END
```

where `Supertype` is a concrete object type, `FieldExtension` is a list of field declarations, and `MethodRevision` is a list of method declarations and method overrides of the form:

```
m := proc
```

where `m` is the name of a method of the supertype and `proc` is a top-level procedure 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 `FieldExtension`, 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 `MethodRevision`, 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.

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.

The names of the fields and methods in an object type declaration must be distinct from one another and from the names of the fields and methods of any of its supertypes.

2.9.4. As an example, 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) = ... ;

  VAR a: A; ab: AB; ...
```

The data record of an `A` object must begin with an `a` field and the data record of an `AB` object must begin with an `a` field and a `b` field. So every `AB` data record is a valid `A` data record, and every `AB` object is also an `A` object.

Since neither `A` nor `AB` have a default value for the `p` method, the method must 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:

---

[2]This restriction is lifted in unsafe modules.

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

creates 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)
```

creates 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 creates 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 create 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 that illustrates the use of default method values:

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

TYPE TextWindow =
  Window OBJECT
    text: Text.T;
    style: TextWindowStyle
  METHODS
    repaint := RepaintTextWindow
  END;
```

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`. These procedures must be declared elsewhere. 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 a reusable queue implementation. First, the interface:

```
TYPE
  Queue = RECORD head, tail: QueueElem END;
  QueueElem = OBJECT link: QueueElem END;

INLINE PROCEDURE Insert(VAR q: Queue; x: QueueElem);
INLINE PROCEDURE Delete(VAR q: Queue): QueueElem;
INLINE PROCEDURE Clear(VAR q: Queue);
```

Then an example client:

```
TYPE
  IntQueueElem = QueueElem OBJECT val: INTEGER END;

VAR
  q: Queue;
  x: IntQueueElem;

...
Clear(q);
NEW(x, 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`.

2.10. Subtyping rules

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

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

`[n..m] <: INTEGER` if `n` and `m` are integers
`[n..m] <: E` if `n` and `m` are from the enumeration type `E`
`[n..m] <: [n'..m']` if `[n..m]` is a (possibly empty) subset of `[n'..m']`

`ARRAY U OF T <: ARRAY V OF T` if `NUMBER(U) = NUMBER(V)`

That is, one fixed array type is a subtype of another if the element types are identical and the index types have the same number of elements.

`ARRAY` I_1 `OF ... ARRAY` I_n `OF T <: (ARRAY OF)`^{n} `T`

if the `I`'s are ordinal types or omitted.

That is, any `n`-dimensional array type with element type `T` is a subtype of `(ARRAY OF)`^{n} `T`. (Omitted `I`'s create open array types. They must precede the fixed array types.)

`SET OF T <: SET OF T'` if `T <: T'`

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

That is, fixed references are subtypes of open references, as long as they are of the same reference class. `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 `(A): Q RAISES E` is covered by `(B): R RAISES F`.

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

`OBJECT ... END <: REFANY`
`UNTRACED OBJECT ... END <: ADDRESS`

That is, every object is a reference.

`NULL <: T OBJECT ... END <: T`

That is, `NIL` is a member of every object subtype (and therefore every object type). 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 identical, 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 the types `AT` and `AU` are unrelated by the subtype relation.

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 either halts (normal outcome), raises an exception, causes a checked runtime error, or loops forever. If the outcome is an exception, we say that the outcome is "tagged" with the exception, together with an argument value if the exception takes an argument.

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

Implementations are expected to speed up normal outcomes at the expense of exceptions (except for the return-exception and exit-exception). Expending ten thousand instructions per exception raised in order 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. It is a checked runtime error (an "unhandled exception") if the outcome of the main program or of any concurrent thread of control is an exception.

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

3.1. Assignment

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

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

`T <: U`, or

`T` is a reference type other than ADDRESS[3] and `U <: T`, 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 length as `v` in each dimension.

The first point can be checked statically; the second generally requires a runtime check.

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

---

[3]This restriction is lifted in unsafe modules.

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 6.2, page 33), the assignment is performed in such a way that no element is used as a target before it is used as a source.

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. 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.

Examples of assignments:

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

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

The same comments would apply if x had an ordinal type and a and b had non-overlapping subtypes, 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 (except in unsafe modules) *)
```

3.2. Procedure call

A procedure call has the form:

```
P(Bindings)
```

where P is a procedure-valued expression and Bindings is a list of *keyword* or *positional* bindings. A keyword binding has the form `name := actual`, where `actual` is an expression and `name` is an identifier. A positional binding has the form `actual`, where `actual` is an expression. When keyword and positional bindings are mixed in a single 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.

It is a static error if the rewritten list of actuals binds any name more than once, binds any name that is not a formal parameter, or fails to bind any formal parameter.

A procedure call can also have the form:

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

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

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

In any call, the actual bound to a READONLY or VALUE formal can be any expression assignable to the type of the formal (except that the prohibition against assigning local procedures is relaxed). The actual bound to a VAR formal must be a writable designator whose type is identical to that of the formal (except that a fixed array actual can be bound to a VAR open array formal if the type of the actual is 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 identical to the type of the formal; otherwise it is treated as a VALUE formal.

In some implementations it may not be possible to take the address of expressions whose types are packed types; implementations are thus free to forbid some or all packed types from being passed as VAR or READONLY parameters.

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 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.

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 supplies no binding for ch. The call P(n := 0, 'a') is illegal, since it has a keyword parameter before a positional parameter.

3.3. EVAL

An EVAL statement has the form:

```
EVAL e
```

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

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

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

3.4. Block statement

A block statement has the form:

```
Decls BEGIN S END
```

where `Decls` is a sequence of declarations and `S` is a statement. The block statement introduces the constants, types, variables, exceptions, and procedures declared in `Decls` and then executes `S`. The scope of the declared names is the block statement. (See Chapter 4, page 25.)

3.5. Sequential composition

A statement of the form:

S_1; S_2

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

3.6. RAISE

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

```
RAISE(e)
```

where `e` is an exception that takes no argument. The outcome of the statement is an exception tagged with `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 an exception tagged with `e` and the value of `x`.

3.7. 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 Handler`$_0$" and each "`(`v_i`)`" are optional. It is a static error for an exception to be named more than once in the list of `id`'s.

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

---

[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.

Body raises any listed exception id_i, then $Handler_i$ is executed. If Body raises any other exception and $Handler_0$ is present, then it is executed. In either case, the outcome of the TRY statement is the outcome of the selected handler. If Body raises an unlisted exception and $Handler_0$ 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 tagged with (id_i, 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.

Any id can be a list of exceptions separated by commas, as shorthand for an expansion in which the rest of the handler is repeated for each exception in the list. That is:

$id_1, \ldots, id_n$ (v) => Handler

is shorthand for:

id_1 (v) => Handler; ...; id_n (v) => Handler

The (v) is optional in this rewriting rule.

3.8. TRY FINALLY

A statement of the form:

TRY S_1 FINALLY S_2 END

executes statement S_1 and then statement S_2. If the outcome of S_1 is normal, the TRY statement is equivalent to S_1; S_2. If S_1 raises an exception and S_2 does not, the exception from S_1 is re-raised after S_2 is executed. If both S_1 and S_2 raise exceptions, the outcome of the TRY is the exception from S_2.

3.9. LOOP

A statement of the form:

```
LOOP S END
```

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

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

3.10. EXIT

A statement of the form:

```
EXIT
```

raises the exit-exception. An EXIT statement that is not textually enclosed by a LOOP, WHILE, REPEAT, or FOR statement is a static error.

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

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

3.11. RETURN

A RETURN statement for a proper procedure has the form:

```
RETURN
```

The statement raises the return-exception without an argument.

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 a static error to include a return value in a proper procedure or for a RETURN statement to occur outside a procedure body. Failure to return a value from a function procedure is a checked runtime error.

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

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

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

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

3.12. IF

An IF statement has the form:

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

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

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

3.13. WHILE

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

```
WHILE B DO S END
```

is shorthand for:

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

3.14. REPEAT

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

```
REPEAT S UNTIL B
```

is shorthand for:

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

3.15. WITH

A WITH statement has the form:

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

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

The statement is 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.

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

```
WITH id1 = e1, id2 = e2, ...
```

is equivalent to:

```
WITH id1 = e1 DO WITH id2 = e2 DO ...
```

3.16. FOR

A FOR statement has the form:

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

where id is an identifier, first and last are expressions whose types are ordinal types with a common supertype, 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. If first and last are integers, the type of id is INTEGER; otherwise the type of id is the enumeration that contains both first and last.

If id is an integer, the statement steps id through the sequence of 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
```

3.17. CASE

A CASE statement has the form:

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

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

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

3.18. TYPECASE

A TYPECASE statement has the form:

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

where Expr is an expression whose type is a reference type, the S's are statements, the T's are reference types, and the v's are identifiers. It is a static error if Expr has type ADDRESS or if any T is not of the same reference class as 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. (A NULL case is useful only if it is the first case.) 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 a series of branches share the same statement and none of them declares a variable, then T_i can be a list of type expressions separated by commas. Such a list is shorthand for an expansion in which the rest of the branch is repeated for each type expression in the list. That is:

$T_1, \ldots, T_n$ => S

is shorthand for:

T_1 => S | ... | T_n => S

For example:

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

3.19. NEW

A NEW statement has the form:

```
NEW(v)
```

where v is a variable whose type is a concrete reference type other than REFANY or ADDRESS. If v has a fixed reference type, the statement sets v to the address of a newly-allocated variable of v's referent type. If v has an object type, the statement sets v to a newly-allocated data record containing the data fields declared in v's type, paired with a method suite containing the methods declared in v's type.

It is a static error if v's referent type is empty. The reference returned by NEW will be distinct from all existing references. The allocated type of the new reference is the type of v.

The initial state of the referent or data record generally represents an arbitrary value of the appropriate type. If v's referent type is an open array type or if v has an object type, then NEW takes additional arguments to control the initial state of the new variable.

A NEW statement for a reference to an open array type has the form:

NEW(v, n_1, ..., n_k)

where the type of v is a concrete reference type whose referent type is a k-dimensional open array type, and the n's are integer-valued expressions. The dimensions of the new array are n_1, ..., n_k. The values in the new array are arbitrary values of the element type.

A NEW statement for an object type has the form:

```
NEW(v, Bindings)
```

where v is a variable whose type is an object type and Bindings is a non-empty list of

positional and keyword bindings used to initialize the new object's fields and methods. The ", Bindings" can be omitted, in which case Bindings is taken to be empty.

Bindings is rewritten to fit the sequence of name-default pairs for the fields and methods of the object type, using the same rewriting rules as for a procedure call (Section 3.2, page 16), except that it is not necessary to provide an initial value for every field. If the bindings are positional, the field bindings precede the method bindings, and within each group, the bindings for a supertype precede the bindings for its subtypes. Fields for which no initial values are provided are initialized to arbitrary values of the appropriate type. It is a static error if any method value is not a top-level procedure constant or has a signature that does not satisfy its method declaration (Section 2.9, page 10).

3.20. 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 Thread.Mutex (page 43). 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 mu or S.

3.21. 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 v or n.

3.22. INCL and EXCL

INCL and EXCL statements have the form:

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

where s designates a variable of a set type and e is an expression assignable to the element type of s. The statements add or remove the element e from the set s, respectively. The statements are equivalent to:

```
WITH x = s DO x := x + T{e} END
WITH x = s DO x := x - T{e} END
```

where T is the type of s and x stands for some variable that does not appear in s or e.

---

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

4. Declarations

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

--- John Scarne's Guide to Modern Poker

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

```
Decls BEGIN S END
```

where `Decls` is a sequence of declarations and `S` is a statement, the executable part of the block. A block can appear as a statement or as the body of a module or procedure. A name can be declared at most once in any block, 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 28). Except for variable initializations, the order of declarations in a block does not matter.

4.1. Types

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

```
TYPE T = U
```

declares `T` to be the type `U`.

4.2. Constants

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

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

declares `id` as a constant whose type is `T` and whose value is 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`.

4.3. Variables

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

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

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

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

```
VAR i: INTEGER := j; j: INTEGER := i; BEGIN S END
```

initializes `i` and `j` to the same arbitrary integer value; it is equivalent to:

```
VAR i: INTEGER; j: INTEGER; 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 an expansion in which the type and initial value are repeated for each identifier in the list. That is:

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

is shorthand for:

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

As a consequence, `E` is evaluated `n` times.

4.4. Procedures

There are two forms of procedure declaration:

```
PROCEDURE id sig = B id

PROCEDURE id sig
```

where `id` is an identifier, `sig` is a procedure signature, and `B` is a block. In both cases, the type of `id` is the procedure type determined by `sig`.

The first form declares `id` to be a procedure constant whose signature is `sig`, body is `B`, and environment is the scope containing the declaration. The scope of the parameter names is `B`. The procedure name `id` must be repeated after the `END` that terminates the body.

The second form is allowed only in interfaces. It 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.

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

4.5. Exceptions

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

```
EXCEPTION id(T)
```

declares `id` as an exception with argument type `T`. If `"(T)"` is omitted, the exception takes no argument.

Any `id` can be a list of identifiers separated by commas, as shorthand for an expansion in which the rest of the declaration is repeated for each identifier in the list. That is:

```
EXCEPTION id1, ..., idn (T)
```

is shorthand for:

```
EXCEPTION id1(T); ...; EXCEPTION idn(T)
```

The `"(T)"` is optional in this rewriting rule.

4.6. Recursive declarations

Recursive declarations are allowed only for constants, types, and procedures. A recursive declaration `N = E` is legal if every occurrence of `N` in `E` is (1) within some occurrence of the type constructor `REF` or `PROCEDURE`, (2) within a field or method type of some occurrence of the type constructor `OBJECT`, or (3) within a procedure body. In case of mutual recursion, the expression `E` is understood to be expanded by replacing names other than `N` with their definitions.

Examples of legal recursive definitions:

```
TYPE
  List = REF RECORD x: REAL; link: List END;
  T = PROCEDURE(r: RECORD n: INTEGER; p: T END)
  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;
```

Examples of illegal recursive definitions:

```
TYPE
  T = RECORD u: U END; U = RECORD t: T END;
  X = X OBJECT END;
CONST
  N = N+1;
VAR v: REF ARRAY [0 .. LAST(v^)] OF INTEGER;
```

4.7. 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 that `T` names some unspecified subtype of `U`. `T` is identified with a particular subtype of `U` elsewhere, by a *type identification*, which has the form:

```
TYPE T == V
```

where `T` is an identifier (possibly qualified by a module name). The type identification `T == V` is allowed only where `T` has been declared as an opaque subtype of some type `U` and `V <: U`. The scope of the type identification is the same as the scope that a declaration would have in the same position. A type identification is not considered a declaration.

If `U` is an object type and `T` is an opaque subtype of `U`, then any variable of type `T` will have all the fields and methods of `U`. Any additional fields and methods of the concrete type will be accessible only in scopes where the type identification is visible.

It is a static error for a program (Section 5.4, page 29) to contain more than one type identification for any opaque type. However, a type identification in an interface can be imported into any scope where it is required. (For an example, see section 5.6, page 30).

An opaque type `T` cannot be used as an object supertype, and variables with an opaque type `T` cannot be allocated with `NEW`, deallocated with `DISPOSE`, or dereferenced. Any other operations allowed on variables of type `U` are allowed.

Note that `TYPECASE` tests the concrete type of a reference at runtime; it is immaterial whether the type name was opaque at compile time. If two opaque types names are identified with the same concrete type, they will be indistinguishable at runtime.

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 that it has different rules 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.

Modules and interfaces are the usual units of compilation. Unlike in Modula-2, there are no nested modules.

5.1. Import statements

There are two forms of IMPORT statement:

```
IMPORT I1, ..., In

FROM I IMPORT N1, ..., Nm
```

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 $I_1, \ldots, I_n$ visible. To refer to an entity named N in any I_j the importer must use the qualified identifier I_j.N.

The second form makes the names $N_1, \ldots, N_m$ from interface I visible. It does not make the name I visible; the importer must refer to N_i and not 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. Both forms make visible any type identifications that are visible in the imported interface. Importing an interface provides access only to the names it declares, not to the names it imports.

5.2. Interfaces

An interface has the form:

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

where id is an identifier that names the interface, Imports is a sequence of import statements, and Decls is a sequence of declarations that contains no procedure bodies or non-constant variable initializations. The names declared in Decls, the visible imported names, and the name id must be distinct.

5.3. Modules

A module has the form:

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

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

If module `M` exports interface `I`, then all declared names in `I` are visible without qualification in `M`.

If module `M` exports interface `I`, then any procedure declared in `I` can be redeclared in `M`, with a body. The signature in the module declaration must be covered by the signature in the interface (as defined in Section 2.8, page 8.) To determine the interpretation of keyword bindings in calls to the procedure, the signature in the implementing module is used within the module; the signature in the interface 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:

```
MODULE M EXPORTS I, J;     INTERFACE I;          INTERFACE 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 that the interface signature covers the module signature. (Within `M`, the signature of `I.X` is determined by the interface declaration and the signature of `X` is determined by the module declaration.)

5.4. Initialization

A program consists of a collection of modules. 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. For example, the main module might be the one named `Main`, or the one that exports some system-dependent interface.

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

5.5. Safety

The keyword UNSAFE can precede the declaration of any interface or module to indicate that it is *unsafe*; that is, that it uses the unsafe features of the language (Chapter 7, page 41). 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.

5.6. 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 New(): T;
END Stack.

MODULE Stack;

  TYPE T == REF RECORD item: REAL; link: T END;

  PROCEDURE Push(VAR s: T; x: REAL) =
    VAR t: T;
    BEGIN
      NEW(t); t.link := s; t.item := x; s := t
    END Push;

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

  PROCEDURE New(): T =
    BEGIN RETURN NIL END New;

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;
  TYPE Stack.T == REF RECORD item: REAL; link: Stack.T END
END StackRep.

MODULE Stack;
  IMPORT StackRep;
  ... Push, Pop, and New 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 consists of 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 for all operations except AND and OR is undefined.

The types of variables, constants, and procedure constants are specified in Chapter 4, Declarations. This chapter specifies the types of other expressions.

6.1. Conventions for describing operations

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

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

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

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 either 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 `ElementType(T)` and `ElementType(a)` are similar.

Keyword parameters cannot be used for arguments to operations.

6.2. 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.

`r^` `r^` denotes the variable addressed by the reference `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]` `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 $a[i_1, \ldots, i_n]$ is shorthand for $a[i_1] \ldots [i_n]$. 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 statically-determined 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, and 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.

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; start, length: CARDINAL): ARRAY OF ElementType(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 `start` elements of `a` and contains the next `length` elements. Note that if `start` 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 for `start + length` to exceed `NUMBER(a)`.

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

---

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

6.3. Numeric literals

Numeric literals denote constant non-negative integers or reals. Integers can be decimal, octal, or hexadecimal; reals are always decimal. The types of these literals are INTEGER, REAL, and LONGREAL.

An literal INTEGER has the form:

```
base_digits
```

where base is either "8", "10", or "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.

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 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, where 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.

Examples:

```
1.    0.5    6.624E-27    3.1415926535d0
```

6.4. Text and character literals

A character literal is a single ASCII 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 enclosed in double quotes. The type of a text literal is Text.T. The following escape sequences can be used to include non-printing characters in text and character literals:

\n	newline (linefeed)	\f	form feed
\t	tab	\\	backslash
\r	carriage return	\"	double quote
\'	single quote		

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

6.5. NIL

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

6.6. Function application

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

6.7. Set, array, and record constructors

A set constructor has the form:

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

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

An array constructor has the form:

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

where A is an array type and the e's are expressions. The constructor denotes a value of type A containing the listed elements in the listed order. The e's must be assignable to the element type of A. As a consequence, 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 e_n can be followed by ". ." to indicate that the value of e_n will be replicated as many times as necessary to fill out the array. It is a static error to provide too many or too few elements for a fixed array type.

A record constructor has the form:

```
R{Bindings}
```

where R is a record type and Bindings is a list of keyword or positional bindings, exactly as in a procedure call (Section 3.2, page 16). 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.

It is a static error if the rewriting binds any field name more than once, binds any name that is not a field of R, or fails to bind any field name of R. Each expression in the binding must be assignable to the type of the corresponding record field.

6.8. 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 44).

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

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

When used as a unary prefix operator, + returns x.

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

When used on sets, + returns the set union of s and t. That is:

```
x IN (s + t)   if and only if   (x IN s) OR (x IN t)
```

The type of s must be a supertype of the type of t, or vice-versa. The type of the result is the supertype.

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

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

When used as a unary prefix operator, - means negation: `- x = 0 - x`.

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

When used on sets, - returns the set difference of `s` and `t`. That is:

`x IN (s - t)` if and only if `(x IN s) AND NOT (x IN t)`

The type of `s` must be a supertype of the type of `t`, or vice-versa. The type of the result is the supertype.

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

A binary infix operator. When used on numeric arguments, * returns the product of `x` and `y`.

When used on sets, * returns the intersection of `s` and `t`. That is:

`x IN (s * t)` if and only if `(x IN s) AND (x IN t)`

The type of `s` must be a supertype of the type of `t`, or vice-versa. The type of the result is the supertype.

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

A binary infix operator. When used on real arguments, / returns the quotient of `x` and `y`.

When used on sets, / returns the symmetric difference of `s` and `t`. That is:

`x IN (s / t)` if and only if `(x IN s) # (x IN t)`

The type of `s` must be a supertype of the type of `t`, or vice-versa. The type of the result is the supertype.

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

`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`.

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

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

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

`ABS` returns the absolute value of `x`.

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

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

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

```
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; it equals FLOOR(r + 0.5). 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, their types must have a common supertype. The type of the result is their smallest common supertype (in the <: order).

6.9. Relations

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

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

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.

```
<=, >=  (x,y: Ordinal)   : BOOLEAN
        (x,y: REAL)      : BOOLEAN
        (x,y: LONGREAL)  : BOOLEAN
        (x,y: ADDRESS)   : BOOLEAN
        (x,y: Set)       : BOOLEAN
```

<= is a binary infix operator. 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.

x >= y is equivalent to y <= x.

```
>, < (x,y: Ordinal)   : BOOLEAN
     (x,y: REAL)      : BOOLEAN
     (x,y: LONGREAL)  : BOOLEAN
     (x,y: ADDRESS)   : BOOLEAN
     (x,y: Set)       : BOOLEAN
```

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

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

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

6.10. Boolean operations

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

NOT p is the complement of p.

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

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

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

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

6.11. Type operations

```
ORD (element: Ordinal): INTEGER
VAL (e: 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 0. 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 e into the element that occupies that position in an enumeration. If T is a subrange, VAL returns the element with the position e in the original enumeration type, not the subrange. It is a checked runtime error for the value of e 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)     : T
      (A: FixedArrayType)  : IndexType(A)
      (a: Array)           : IndexType(a)

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

For an ordinal type `T`, `FIRST` returns the smallest value of `T` and `LAST` returns the largest value. If `T` is an empty subrange of the integers, `FIRST` returns `0` and `LAST` returns `-1`. If `T` is an empty enumeration, `FIRST(T)` and `LAST(T)` are static errors.

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))`. In this case, 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`.

```
TYPECODE (T: RefType) : INTEGER
         (r: REFANY)  : INTEGER
```

Every reference type has an associated integer code. Different types have different codes. `TYPECODE(T)` returns the code for the type `T` and `TYPECODE(r)` returns the code for the allocated type of `r`. The code for a type is constant for any single execution of a program, but may be different for different executions.

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

`NARROW` checks that `x` is a member of `T` and returns the value of `x`. If the check fails, a runtime error occurs. The type of `NARROW(x, T)` is `T`. It is a static error unless `T` is a traced reference or object type and `x` is assignable to `T`.

```
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` returns the number of 8-bit bytes, and `ADRSIZE` returns the number of addressable locations.

In all cases, `x` must be a designator and `T` must not be an open array type. A designator `x` will be evaluated only if its type is an open array type.

6.12. Text operations

```
& (a,b: Text.T): Text.T
```

`a & b` is the concatenation of `a` and `b`, as defined by `Text.Cat`. (Page 42.)

6.13. 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. The following operations are legal in constant expressions:

```
+  -  /  *  &

=  #  <  >  <=  >=

[ ]   .

{ }   (set, array, and record constructors)
```

ABS	DIV	LONGFLOAT	NOT	TRUNC
ADRSIZE	FIRST	LOOPHOLE	NUMBER	VAL
AND	FLOAT	MAX	OR	
BITSIZE	FLOOR	MIN	ORD	
BYTESIZE	IN	MOD	ROUND	
CEILING	LAST	NARROW	SUBARRAY	

All required operations in the Text, Word, and Fmt interfaces (Chapter 8, page 42) are allowed in constant expressions.

The following operations are not legal in constant expressions:

```
ADR     ^     TYPECODE     function application
```

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.

`NIL` is the only possible value of a procedure or reference constant expression.

6.14. Precedence

Modula-3 operations include infix, prefix, postfix, and applicative operators. All operators are left associative; parentheses can be used to override the precedence rules. Here is the list of operators, in order of decreasing binding power:

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

Here are some examples of expressions together with their fully parenthesized forms:

```
M.F(x)              (M.F)(x)              . before application
M.F(x)^             (M.F(x))^             calls can be dereferenced
- p^                - (p^)                dereference before unary minus
a + b MOD c         a + (b MOD c)         MOD before +
x IN s + t          x IN (s + t)          + before IN
NOT x < y AND p     (NOT (x < y)) AND p   < before NOT before AND
A OR B AND C        A OR (B AND C)        AND before OR
```

In a procedure type, : binds tighter than `RAISES`. That is, the parentheses are required in

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

7. Unsafe operations

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

--- Aristotle, Nicomachean Ethics

The features defined in this chapter can potentially cause unchecked runtime errors and are thus generally forbidden in safe modules. An implementation can allow restricted use of these features in safe modules if it can guarantee that no unchecked errors will result.

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` returns the address of the variable `x`. 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 the type of `v` is a concrete 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`.

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, Word, and Fmt as specified in this chapter. Implementations are free to extend these interfaces, as long as they do not invalidate clients of the unextended interfaces.

If a shared variable is written concurrently by two threads, or written by one and read concurrently by another, the effect is implementation-dependent. For example, if a thread writes two shared variables, another thread that reads them concurrently might see the second update before seeing the first. Consequently, to write portable concurrent programs, the Thread interface must be used to provide mutual exclusion for shared variables. The implementation of the Thread interface is required to satisfy the specification in "Synchronization Primitives for a Multiprocessor: A Formal Specification" [1].

For a semantic specification of an extended version of the Text interface, see "The Modula-2+ Text Interface" [4].

8.1. The Text interface

```
INTERFACE Text;

TYPE
  T <: REFANY;
  (* A sequence of characters.  The index of the first is 0. *)

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

PROCEDURE Equal(t, u: T): BOOLEAN;
(* Return TRUE if t and u have the same length and the same
   (case-sensitive) contents. *)

PROCEDURE GetChar(t: T; i: CARDINAL): CHAR;
(* Return character i of t.  A checked error if i >= Length(t). *)

PROCEDURE Length(t: T): CARDINAL;
(* Return the number of characters in t. *)

PROCEDURE Sub(t: T; start, length: CARDINAL): T;
(* Returns a sub-sequence of t.  The result will be empty if
   start >= Length(t); otherwise the range of indexes of the
   subsequence is [start .. MIN(length, Length(t) - start)]. *)

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

PROCEDURE FromStr(VAR IN a: ARRAY OF CHAR): Text.T;
(* Return a text containing the characters of a. *)

END Text.
```

8.2. The Thread interface

```
INTERFACE Thread;

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

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

PROCEDURE InitMutex(VAR m: Mutex);
(* To be called once before m is used.  Leaves m unlocked. *)

PROCEDURE InitCondition(VAR c: Condition);
(* To be called once before c is used.  Leaves c with no waiting threads. *)

PROCEDURE Fork(cl: Closure): T;
(* Return a handle on a newly-created thread that executes cl.apply(). *)

PROCEDURE Join(t: T): REFANY;
(* Wait until t has terminated and return its result.  It is a
   checked runtime 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 waits for m to be unlocked, locks it, 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 cease waiting and become eligible to run.  *)

PROCEDURE Signal(c: Condition);
(* One or more threads waiting on c cease waiting and become
   eligible to run.  A no-op if no threads are waiting on c. *)

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

EXCEPTION Alerted; (* For approximating asynchronous interrupts *)

PROCEDURE Alert(t: T);
(* Mark t as an alerted thread. *)

PROCEDURE TestAlert(): BOOLEAN;
(* Returns 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 the thread is marked alerted at the time of
   call or sometime during the wait, raise Alerted.  Implementations
   may raise additional exceptions. *)

END Thread.
```

8.3. The Word interface

```
INTERFACE Word;

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

        w_0, ..., w_Word.Size-1.

   It also represents the unsigned number

       sum of 2^i * w_i for i in 0, ..., Word.Size-1. *)

TYPE T = INTEGER;
(* encoding is implementation-dependent; e.g., 2's complement. *)

CONST Size = ...;                    (* implementation-dependent *)

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 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 dividing by 2^n *)

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

PROCEDURE Extract(x: T; i, n: CARDINAL): T;
(* Take 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(VAR x: T; y: T; i, n: CARDINAL): T;
(* Replace 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.
```

8.4. The Fmt interface

```
INTERFACE Fmt;
IMPORT Text;

TYPE
  Align = {Left, Right};
  Base = {Oct, Dec, Hex};  (* Number base can be 8, 10, or 16 *)
  Style = {Flo, AltFlo, Sci, AltSci, Mix};
    (* Formatting styles for REALs.  The Sci and AltSci formats are
       "decimal E exponent"; the Flo and AltFlo formats are simply
       "decimal".  In the Alt formats, trailing zeros are suppressed
       in the decimal part; in both formats, a decimal point is always
       printed.  The Mix format is AltFlo unless AltSci is shorter;
       if AltFlo is selected and there are no zeros after the decimal
       point, the decimal point is suppressed. *)

PROCEDURE Bool(b: BOOLEAN): Text.T;
(* Format b as "TRUE" or "FALSE". *)

PROCEDURE Int(n: INTEGER; base: Base := Dec): Text.T;
(* Format n in the given base. *)

PROCEDURE Addr(n: ADDRESS; base: Base := Hex): Text.T;
(* Format n in the given base.  Return "NIL" if n = NIL. *)

PROCEDURE Ref(r: REFANY; base: Base := Hex): Text.T;
(* Format r in the given base.  Return "NIL" if n = NIL. *)

PROCEDURE Real(
    x: REAL;
    precision: CARDINAL := 6;
    style: Style := Mix)
    : Text.T;
(* Format x in the given style.  The precision is the number of fractional
   digits in the decimal, or the maximum number for the Alt formats.  *)

PROCEDURE LongReal(
    x: LONGREAL;
    precision: CARDINAL := 6;
    style: Style := Mix)
    : Text.T;
(* Format x in the given style.  The precision is the number of fractional
   digits in the decimal, or the maximum number for the Alt formats.  *)

PROCEDURE Char(c: CHAR): Text.T;
(* Return a text containing the character c. *)

PROCEDURE Pad(
    text: Text.T;
    length: CARDINAL;
    padChar: CHAR := ' ';
    align: Align := Right)
    : Text.T;
(* If Text.Length(text) <= length, then text is returned unchanged.
   Otherwise, text is padded with padChar until it has the given length.
   The text goes to the right or left, according to align. *)

END Fmt.
```

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

9.1. Keywords

Each Modula-3 keyword is recognized in two forms: all upper case and all lower case. Here are the upper case forms:

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

9.2. Identifiers

A *reserved* identifier cannot be redeclared. Each reserved identifier is recognized in two forms: all upper case and all lower case. Here are the upper case forms:

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

9.3. Operators

The following characters and character pairs are classified as operators:

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

9.4. Comments

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

9.5. 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 in Section 9.10, page 48. Each production is terminated by a period. Indented productions are used only in the productions immediately above them.

9.6. Compilation unit productions

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

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

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

9.7. Statement productions

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

  Stmt =  AssignmentStmt | Block | CallStmt | CaseStmt | ExitStmt | EvalStmt
        | ForStmt | IfStmt | LockStmt | LoopStmt | RepeatStmt | ReturnStmt
        | TryFinallyStmt | TryStmt | TypeCaseStmt | WhileStmt | WithStmt.

AssignmentStmt = 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.
RepeatStmt     = REPEAT Stmts UNTIL Expr.
ReturnStmt     = RETURN [ Expr ].
TryFinallyStmt = TRY Stmts FINALLY Stmts END.
TryStmt        = TRY Stmts EXCEPT [Handler] {"|" Handler} [ELSE Stmts] END.
TypeCaseStmt   = TYPECASE Expr OF [ Tcase ] { "|" Tcase } [ELSE 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 ).
```

9.8. Type productions

```
Type = TypeID | ArrayType | PackedType | EnumerationType | ObjectType
     | ProcedureType | RecordType | RefType | SetType | SubrangeType
     | "(" Type ")".

ArrayType       =  ARRAY [ Type { "," Type } ] OF Type.
PackedType      =  BITS ConstExpr FOR Type.
EnumerationType =  "{" [ IDList ] "}".
ObjectType      =  [ Type | UNTRACED ] OBJECT Fields [ METHODS Methods ] END.
ProcedureType   =  PROCEDURE Signature.
RecordType      =  RECORD Fields END.
RefType         =  [ UNTRACED ] REF Type.
SetType         =  SET OF Type.
SubrangeType    =  "[" ConstExpr ".." ConstExpr "]".

  Fields  = [ Field { ";" Field } [ ";" ] ].
  Field   = IDList ( ":" Type & ":=" ConstExpr ).
  Methods = [ Method { ";" Method } [ ";" ] ].
  Method  = Ident ( Signature & ":=" ProcedureID ).
```

9.9. 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 } [ "," ".." ] .
```

9.10. Miscellaneous productions

```
TypeID      =  Ident [ "." Ident ].
ExceptionID =  Ident [ "." Ident ].
ProcedureID =  Ident [ "." Ident ].

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

9.11. Token productions

To read a token, first skip all blanks, tabs, newlines, carriage returns, vertical tabs, form feeds, and comments. Then read the longest sequence of characters that forms an operator (as defined in Section 9.3, page 46) 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 9.1, a reserved identifier if it appears in Section 9.2, and an ordinary identifier otherwise.

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

```
Literal = Number | CharLiteral | TextLiteral.

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

Number = Digit { Digit }
       | Digit { Digit } "_" HexDigit { HexDigit }
       | Digit { Digit } "." Digit { Digit } [ Exponent ].

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

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

TextLiteral = DQUOTE { Character | Escape } DQUOTE.

Character = Letter | Digit | OtherChar.

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

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

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

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

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

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

`OtherChar` consists of all printing ASCII characters except double-quote, right single quote, and backslash.

References

[1] A.D. Birrell, J.V. Guttag, J.J. Horning, 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] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard.
Simula Begin.
Auerbach, Philadelphia PA, 1973.

[3] C.A.R. Hoare.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10), October 1974.

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

[5] Butler W. Lampson.
A Description of the Cedar Language.
Technical Report CSL-83-15, Xerox Palo Alto Research Center, December 1983.

[6] 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.

[7] Butler W. Lampson and David D. Redell.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2), February 1980.

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

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

[10] Paul Rovner.
Extending Modula-2 to Build Large, Integrated Systems.
IEEE Software 3(6), November 1986.

[11] Larry Tesler, Apple Computers.
Object Pascal Report.
Structured Language World 9(3), 1985.

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

[13] N. Wirth.
From Modula to Oberon and The Programming Language Oberon.
Technical Report 82, Institut fur Informatik, ETH Zurich, September 1987.

Index

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating go to's while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986. Revised January 26, 1988

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

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

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301