113

Some Useful Modula-3 Interfaces

Jim Horning, Bill Kalsow, Paul McJones, Greg Nelson

December 25, 1993

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Some Useful Modula-3 Interfaces

Jim Horning, Bill Kalsow, Paul McJones, Greg Nelson

December 25, 1993

Authors' Abstract

This manual describes a collection of interfaces defining abstractions that SRC's programmers have found useful over a number of years of experience with Modula-3 and its precursors. We hope the interfaces will be useful as a "starter kit" of abstractions, and as a model for designing and specifying abstractions in Modula-3.

Contents

Acknowledgments

This manual builds on, and incorporates, the work of many people. We've borrowed many ideas from the literature, as documented in the references. But many other people have helped us design, implement, and refine the specific interfaces published in this manual:

Lyle Ramshaw orchestrated a redesign of the `Fmt` and `Lex` interfaces and the addition of the `ToDecimal` and `FromDecimal` procedures to the `Float` generic interface. David Goldberg and Jorge Stolfi consulted. Luca Cardelli designed the first version of `Lex` (which he called `Sift`).

`Atom`, `List`, `Sx`, and `Table` are similar to Modula-2+ versions designed by John Ellis. Jim Meehan collaborated on the design of `Sx`. Eric Muller and Jorge Stolfi explored the use of object types and generic interfaces for abstractions such as tables.

`Wr` and `Rd` were designed by Mark R. Brown and Greg Nelson and closely follow the design of the Modula-2+ streams package.

`Pathname` borrows from an earlier version written by Eric Muller.

Jim Horning and John Guttag wrote formal Larch specifications of `File.T` and its subtypes, and in the process asked questions leading to substantial improvements to the specifications in this manual.

Mick Jordan helped refine the operating-system interfaces and wrote the first implementations of them.

`WeakRef` was greatly improved through long and spirited discussions involving Luca Cardelli, Dave Detlefs, John Ellis, John DeTreville, Mick Jordan, Bill Kalsow, Mark Manasse, Ted Wobber, and Garret Swart. John DeTreville provided the implementation.

John DeTreville collaborated on the design of `RTCollector` and the other runtime interfaces.

Mary-Claire van Leunen encouraged us to begin.

Finally, this manual was greatly improved by the group of people who read earlier drafts and participated in a running discussion on an electronic bulletin board: Ed Balkovich, Andrew Birrell, Hans Boehm, Marc H. Brown, Mike Burrows, Luca Cardelli, Michel Dagenais, Alan Demers, Dave Detlefs, Mike Dixon, John DeTreville, Steve Freeman, Michel Gangnet, David Goldberg, Judy Hall, Dave Hanson, Carl Hauser, Allan Heydon, Chuck Jerian, Butler Lampson, Mark Manasse, Eric Muller, Hal Murray, David Nichols, Sharon Perl, Dave Redell, Eric Roberts, Robert Sedgewick, Mike Spreitzer, Garret Swart, Samuel Weber, and Ted Wobber.

1 Introduction

Modula-3 invites you to structure your program as a set of modules interconnected via interfaces. Each interface typically corresponds to an abstract data type. Some of these abstractions are particular to the program at hand, but others are more general. This manual describes a collection of interfaces defining abstractions that SRC's programmers have found useful over a number of years of experience with Modula-3 and its precursors.

This manual concentrates on basic abstractions such as the standard interfaces required or recommended by the Modula-3 language definition, various data structures, portable operating-system functions, and control of the Modula-3 runtime. For building distributed systems, see [2]. For building user interfaces, see [11], [4], and [5].

1.1 Naming conventions for types.

We generally give the name `T` to the main type in an interface. For example, the main type in the `Date` interface is `Date.T`.

Most object types have a method that is responsible for initializing the object. By convention, this method is named `init`, and returns the object after initializing it, so that the object can be initialized and used in an expression at the same time: for example,

```
VAR s := NEW(Sequence.T).init();
```

If there are several different ways to initialize the object, there will be several methods. The most basic will be named `init` and the others will receive descriptive names. For example, `Sequence.T.init` initializes an empty sequence; `Sequence.T.fromArray` initializes a sequence from an array.

Many of our types are "abstract" in the sense that they define the methods of a type, but not their implementations. Various subtypes of the abstract type define different methods corresponding to different instances of the abstract type. For example, the type `Rd.T` is a abstract reader (a stream of input characters). Its subtype `FileRd.T` is a reader whose source is a file; its subtype `TextRd.T` is a reader whose source is a text string.

If you allocate an object of an abstract type and attempt to use it, you will almost certainly get a checked runtime error, since its methods will be `NIL`. Therefore, you must read the interfaces to find out which types are abstract and which are concrete. The typical pattern is that an abstract type does not have an `init` method, but each of its concrete instances does. This allows different subtypes to be initialized differently. For example, `FileRd.T` has an `init` method that takes a file; `TextRd.T` has an `init` method that takes a text; and `Rd.T` has no `init` method at all.

For some abstract types we choose to honor one of its subtypes as a "default implementation". For example, we provide a hash table implementation as the default for the abstract type `Table.T`. In this case we vary the naming convention: instead of a separate interface `HashTable` defining the concrete type `HashTable.T` as a subtype of `Table.T`, we declare the default concrete type in the same interface with the abstract type and give it the name `Default`. Thus `Table.T` and `Table.Default` are respectively the abstract table type and its default implementation via hash tables. If you want to allocate a table you must allocate a `Table.Default`, not a `Table.T`. On the other hand, if you are defining a procedure that requires a table as a parameter, you probably want to declare the parameter as a `Table.T`, not a `Table.Default`, to avoid excluding other table implementations.

We use abstract types only when they seem advantageous. Thus the type `Sequence.T`, which represents an extensible sequence, could have been an abstract type, since different implementations are easy to imagine. But engineering considerations argue against multiple implementations, so we declared `Sequence.T` as a concrete type.

1.2 Concurrency.

The specification of a Modula-3 interface must explain how to use the interface in a multithreaded program. When not otherwise specified, each procedure or method is *atomic*: it transforms an initial state to a final state with no intermediate states that can be observed by other threads.

Alternatively, a data structure (the state of an entire interface, or of a particular instance of an object type) can be specified as *unmonitored*, in which case the procedures and methods operating on it are not necessarily atomic. In this case it is the client's responsibility to ensure that multiple threads are not accessing the data structure at the same time—or more precisely, that this happens only if all the concurrent accesses are read-only operations. Thus for an unmonitored data structure, the specification must state which procedures or methods are read-only.

If all operations are read-only, there is no difference between monitored and unmonitored data structures.

1.3 Aliasing.

The procedures and methods defined in this manual are not guaranteed to work with aliased `VAR` parameters.

1.4 Exception parameters for abstract types.

It is often useful for an exception to include a parameter providing debugging information of use to the programmer, especially when the exception signals abstraction failure. Different implementations of an abstract type may wish to supply different debugging information. By convention, we use the type `AtomList.T` for this purpose. The first element of the list is an error code; the specification of the subsequent elements is deferred to the subtypes. Portable modules should treat the entire parameter as an opaque type.

An implementation module can minimize the probability of collision by prefixing its module name to each atom that it includes in the list.

1.5 Standard generic instances.

Several of the interfaces in this manual are generic. Unless otherwise specified, standard instances of these interfaces are provided for all meaningful combinations of the formal imports ranging over `Atom`, `Integer`, `Refany`, and `Text`.

For each interface that is likely to be used as a generic parameter, we define procedures `Equal`, `Compare`, and `Hash`.

The procedure `Equal` must compute an equivalence relation on the values of the type; for example, `Text.Equal(t, s)` tests whether `t` and `s` represent the same string. (This is different from `t = s`, which tests whether `t` and `s` are the same reference.)

If there is a natural total order on a type, then we define a `Compare` procedure to compute it, as follows:

```
PROCEDURE Compare(x, y: X): [-1..1];
```

Return

> *-1* if *x R y* and not *Equal(x, y)*,
> *0* if *Equal(x, y)*, and
> *1* if *y R x* and not *Equal(x, y)*.

(Technically, `Compare` represents a total order on the equivalence classes of the type with respect to `Equal`.) If there is no natural order, we define a `Compare` procedure that causes a checked runtime error. This allows you to instantiate generic routines that require an order (such as sorting routines), but requires you to pass a compare procedure as an explicit argument when calling the generic routine.

The function `Hash` is a hash function mapping values of a type `T` to values of type `Word.T`. This means that (1) it is time-invariant, (2) if `t1` and `t2` are values of type `T` such that `Equal(t1, t2)`, then `Hash(t1) = Hash(t2)`, and (3) its range is distributed uniformly throughout `Word.T`.

Note that it is not valid to use `LOOPHOLE(r, INTEGER)` as a hash function for a reference `r`, since this is not time-invariant on implementations that use copying garbage collectors.

1.6 Sets and relations.

The specifications in this manual are written informally but precisely, using basic mathematical concepts. For completeness, here are definitions of these concepts.

A *set* is a collection of elements, without consideration of ordering or duplication: two sets are equal if and only if they contain the same elements.

If `X` and `Y` are sets, a *map* `m` from `X` to `Y` uniquely determines for each `x` in `X` an element `y` in `Y`; we write `y = m(x)`. We refer to the set `X` as the *domain* of `m`, or `dom(m)` for short, and the set `Y` as the *range* of `m`. A *partial* map from `X` to `Y` is a map from some subset of `X` to `Y`.

If `X` is a set, a *relation* `R` on `X` is a set of ordered pairs `(x, y)` with `x` and `y` elements of `X`. We write `x R y` if `(x, y)` is an element of `R`.

A relation `R` on `X` is *reflexive* if `x R x` for every `x` in `X`; it is *symmetric* if `x R y` implies that `y R x` for every `x`, `y` in `X`; it is *transitive* if `x R y` and `y R z` imply `x R z` for every `x`, `y`, `z` in `X`; and it is an *equivalence relation* if it is reflexive, symmetric, and transitive.

A relation `R` on `X` is *antisymmetric* if for every `x` and `y` in `X`, `x = y` whenever both `x R y` and `y R x`; `R` is a *total order* if it is reflexive, antisymmetric, transitive, and if, for every `x` and `y` in `X`, either `x R y` or `y R x`.

If `x` and `y` are elements of a set `X` that is totally ordered by a relation `R`, we define the *interval* `[x..y]` as the set of all `z` in `X` such that `x R z` and `z R y`. Note that the notation doesn't mention `R`, which is usually clear from the context (e.g., $\leq$ for numbers). We say `[x..y]` is *closed* at its upper and lower endpoints because it includes `x` and `y`. Half-open and open intervals exclude one or both endpoints; notationally we substitute a parenthesis for the corresponding bracket, for example `[x..y)` or `(x..y)`.

A *sequence* `s` is a map whose domain is a set of consecutive integers. In other words, if `dom(s)` is not empty, there are integers `l` and `u`, with `l<=u`, such that `dom(s)` is `[l..u]`. We often write `s[i]` instead of `s(i)`, to emphasize the similarity to a Modula-3 array. If the range of `s` is `Y`, we refer to `s` as a sequence of `Y`'s. The *length* of a sequence `s`, or `len(s)`, is the number of elements in `dom(s)`.

In the specifications, we often speak of assigning to an element of a sequence or map, which is really a shorthand for replacing the sequence or map with a suitable new one. That is, assigning `m(i) := x` is like assigning `m := m'`, where `dom(m')` is the union of `dom(m)` and `{i}`, where `m'(i) = x`, and where `m'(j) = m(j)` for all `j` different from `i` and in `dom(m)`.

If `s` is a finite sequence, and `R` is a total order on the range of `s`, then *sorting* `s` means to reorder its elements so that for every pair of indexes `i` and `j` in `dom(s)`, `s[i] R s[j]` whenever `i <= j`. We say that a particular sorting algorithm is *stable* if it preserves the original order of elements that are equivalent under `R`.

2 Standard interfaces

This section presents the interfaces required by every Modula-3 implementation. The versions included here have some minor changes and additions to the versions in [13].

`Text` provides operations on text strings.

`Thread` provides synchronization primitives for multiple threads of control.

`Word` provides operations on unsigned words.

`Real`, `LongReal`, and `ExtendedReal` define the properties of the three floating-point types; for example, their bases and ranges.

`RealFloat`, `LongRealFloat`, and `ExtendedFloat` provide numerical operations related to the floating-point representation; for example, extracting the exponent of a number.

`FloatMode` provides operations for testing (and possibly setting) the behavior of the implementation in response to numeric conditions; for example, overflow.

This section also presents two related interfaces provided by SRC Modula-3 and recommended to other implementers, but not required:

`Lex` provides for parsing numbers and other data from an input stream.

`Fmt` provides for textual formatting of numbers and other data.

2.1 Text

A non-nil `TEXT` represents an immutable, zero-based sequence of characters. `NIL` does not represent any sequence of characters, it will not be returned from any procedure in this interface, and it is a checked runtime error to pass `NIL` to any procedure in this interface.

```
INTERFACE Text;

IMPORT Word;

TYPE T = TEXT;

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 (case-sensitive) contents.*

```
PROCEDURE GetChar(t: T; i: CARDINAL): CHAR;
```
Return character `i` *of* `t`*. It is a checked runtime error if* `i >= Length(t)`*.*

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

```
PROCEDURE Empty(t: T): BOOLEAN;
```
Equivalent to `Length(t) = 0`*.*

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

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

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

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

```
PROCEDURE Hash(t: T): Word.T;
```
Return a hash function of the contents of `t`*.*

```
PROCEDURE Compare(t1, t2: T): [-1..1];
```
Return -1 if `t1` *occurs before* `t2`*, 0 if* `Equal(t1, t2)`*, +1 if* `t1` *occurs after* `t2` *in lexicographic order.*

```
PROCEDURE FindChar(t: T; c: CHAR; start := 0): INTEGER;
```
If `c = t[i]` *for some* `i` *in* `[start .. Length(t)-1]`*, return the smallest such* `i`*; otherwise, return -1.*

```
PROCEDURE FindCharR(t: T; c: CHAR;
  start := LAST(INTEGER)-5): INTEGER;
```
If `c = t[i]` *for some* `i` *in* `[0 .. MIN(start, Length(t)-1)]`*, return the largest such* `i`*; otherwise, return -1.*

```
END Text.
```

SRC Implementation Note. The default value for the `start` parameter of `FindCharR` was chosen to avoid a bug in some C compilers.

2.2 Thread

A `Thread.T` is a handle on a thread. A `Mutex` is locked by some thread, or unlocked. A `Condition` is a set of waiting threads. A newly-allocated `Mutex` is unlocked; a newly-allocated `Condition` is empty. It is a checked runtime error to pass the `NIL` `Mutex`, `Condition`, or `T` to any procedure in this interface.

The `Thread` interface is based on Hoare's monitors [7], as modified in Mesa [10] and simplified in Modula-2+ [15]. For a tutorial on threads and a formal specification of the interface, see chapters 4 and 5 of [13].

```
INTERFACE Thread;

TYPE
  T <: ROOT;
  Mutex = MUTEX;
  Condition <: ROOT;

TYPE Closure = OBJECT METHODS apply(): REFANY END;

PROCEDURE Fork(cl: Closure): T;
```

Return a handle on a newly created thread executing `cl.apply()`.

```
PROCEDURE Join(t: T): REFANY;
```

Wait until `t` has terminated and return its result. It is a checked 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 unlock `m` and wait on `c`. Then relock `m` and returns.

```
PROCEDURE Acquire(m: Mutex);
```

Wait until `m` is unlocked and then lock it.

```
PROCEDURE Release(m: Mutex);
```

The calling thread must have `m` locked. Unlock `m`.

```
PROCEDURE Broadcast(c: Condition);
```

Make all threads waiting on `c` eligible to run.

```
PROCEDURE Signal(c: Condition);
```

One or more threads waiting on `c` *become eligible to run.*

`PROCEDURE Pause(n: LONGREAL);`

Wait for `n` *seconds to elapse.*

To wait until a specified point in time in the future, say `t`, you can use the call

```
Pause(t - Time.Now())
```

`PROCEDURE Self(): T;`

Return the handle of the calling thread.

`EXCEPTION Alerted;`

Used to approximate asynchronous interrupts.

`PROCEDURE Alert(t: T);`

Mark `t` *as an alerted thread.*

`PROCEDURE TestAlert(): BOOLEAN;`

If the calling thread has been marked alerted, return `TRUE` *and unmark it.*

`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`*.*

`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`*.*

`PROCEDURE AlertPause(n: LONGREAL) RAISES {Alerted};`

Like `Pause`*, but if the thread is marked alerted at the time of the call or sometime during the wait, raise* `Alerted`*.*

`CONSTANT AtomicSize = ...;`

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

`END Thread.`

2.3 Word

A `Word.T w` represents a sequence of `Word.Size` bits $w_0, \ldots, w_{\texttt{Word.Size-1}}$. It also represents the unsigned number $\sum_i w_i \cdot 2^i$. Finally, it also represents a signed `INTEGER` by some implementation-dependent encoding (for example, two's complement). The built-in operations of the language deal with the signed value; the operations in this interface deal with the unsigned value or with the bit sequence.

```
INTERFACE Word;

TYPE T = INTEGER;

CONST Size: INTEGER = BITSIZE(T);
```

Here are the arithmetic operations on unsigned words:

```
PROCEDURE Plus(x, y: T): T;     (* (x + y) MOD 2^Word.Size *)
PROCEDURE Times(x, y: T): T;    (* (x * y) MOD 2^Word.Size *)
PROCEDURE Minus(x, y: T): T;    (* (x - y) MOD 2^Word.Size *)
PROCEDURE Divide(x, y: T): T;   (* x DIV y *)
PROCEDURE Mod(x, y: T): T;      (* x MOD y *)
PROCEDURE LT(x, y: T): BOOLEAN; (* x < y *)
PROCEDURE LE(x, y: T): BOOLEAN; (* x <= y *)
PROCEDURE GT(x, y: T): BOOLEAN; (* x > y *)
PROCEDURE GE(x, y: T): BOOLEAN; (* x >= y *)
```

And here are the logical operations on bit sequences:

```
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 *)
```

And here are additional operations on bit sequences:

```
PROCEDURE Shift(x: T; n: INTEGER): T;
```

For all `i` such that both `i` and `i - n` are in the range `[0..Word.Size - 1]`, bit `i` of the result equals bit `i - n` of `x`. The other bits of the result are 0. Thus shifting by `n > 0` is like multiplying by 2^n.

Since Modula-3 has no exponentiation operator, `Word.Shift(1, n)` is the usual way of writing 2^n in a constant expression.

```
PROCEDURE LeftShift(x: T; n: [0..Size-1]): T;
```

Equivalent to `Shift(x, n)`.

```
PROCEDURE RightShift(x: T; n: [0..Size-1]): T;
```
Equivalent to `Shift(x, -n)`.

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

```
PROCEDURE LeftRotate(x: T; n: [0..Size-1]): T;
```
Equivalent to `Rotate(x, n)`.

```
PROCEDURE RightRotate(x: T; n: [0..Size-1]): T;
```
Equivalent to `Rotate(x, -n)`.

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

```
PROCEDURE Insert(x, y: T; i, n: CARDINAL): T;
```
Result of replaceing `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.
```

2.4 Real, LongReal, and Extended

For definitions of the terms used in the floating-point interfaces, see the ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

The interfaces `Real`, `LongReal`, and `Extended` define constant attributes of the three built-in floating-point types:

```
INTERFACE Real; TYPE T = REAL;
CONST
  Base: INTEGER = ...;
  Precision: INTEGER = ...;
  MaxFinite: T = ...;
  MinPos: T = ...;
  MinPosNormal: T = ...;
  MaxExpDigits: INTEGER = ...;
  MaxSignifDigits: INTEGER = ...;
END Real.
```

```
INTERFACE LongReal; TYPE T = LONGREAL;
CONST
  Base: INTEGER = ...;
  Precision: INTEGER = ...;
  MaxFinite: T = ...;
  MinPos: T = ...;
  MinPosNormal: T = ...;
  MaxExpDigits: INTEGER = ...;
  MaxSignifDigits: INTEGER = ...;
END LongReal.
```

```
INTERFACE Extended; TYPE T = EXTENDED;
CONST
  Base: INTEGER = ...;
  Precision: INTEGER = ...;
  MaxFinite: T = ...;
  MinPos: T = ...;
  MinPosNormal: T = ...;
  MaxExpDigits: INTEGER = ...;
  MaxSignifDigits: INTEGER = ...;
END Extended.
```

The specification is the same for all three interfaces:

> `Base` is the base of the floating-point representation for `T`.
>
> `Precision` is the number of base-`Base` digits of precision for `T`.
>
> `MaxFinite` is the maximum finite value in `T`. For non-IEEE implementations, this is the same as `LAST(T)`.
>
> `MinPos` is the minimum positive value in `T`.
>
> `MinPosNormal` is the minimum positive normal value in `T`; it differs from `MinPos` only for implementations (like IEEE) with denormalized numbers.
>
> `MaxExpDigits` is the smallest integer with the property that every finite number of type `T` can be written in base-10 scientific notation using an exponent with at most `MaxExpDigits`.
>
> `MaxSignifDigits` is the smallest integer with the property that for every point `x` along the number line, the two floating-decimal numbers with `MaxSignifDigits` closest to `x` are closer to each other than are the two closest numbers of type `T`.

Typically,

```
MaxExpDigits    = CEILING(LOG_10(LOG_10(MaxFinite)))
MaxSignifDigits = CEILING(LOG_10(Base^Precision)) + 1.
```

2.5 RealFloat, LongRealFloat, and ExtendedFloat

The interfaces `RealFloat`, `LongRealFloat`, and `ExtendedFloat` define operations that depend on the floating-point representation. Each one is an instance of a generic interface `Float`:

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

The generic interface `Float` provides access to the floating-point operations required or recommended by the IEEE floating-point standard. Consult the standard to resolve any fine points in the specification of the procedures. Non-IEEE implementations that have values similar to NaNs and infinities should explain how those values behave in an implementation guide. (NaN is an IEEE term whose informal meaning is "not a number".)

```
GENERIC INTERFACE Float(R);

IMPORT FloatMode;

TYPE T = R.T;

PROCEDURE Scalb(x: T; n: INTEGER): T RAISES {FloatMode.Trap};
```

Return `x` $\cdot\, 2^{\mathtt{n}}$.

```
PROCEDURE Logb(x: T): T RAISES {FloatMode.Trap};
```

Return the exponent of `x`. *More precisely, return the unique integer n such that the ratio* `ABS(x) / Base`^{n} *is in the half-open interval* `[1..Base)`, *unless* `x` *is denormalized, in which case return the minimum exponent value for* `T`.

```
PROCEDURE ILogb(x: T): INTEGER;
```

Like `Logb`, *but returns an integer, never raises an exception, and always returns the n such that* `ABS(x) / Base`^{n} *is in the half-open interval* `[1..Base)`, *even for denormalized numbers. Special cases: it returns* `FIRST(INTEGER)` *when* `x` *= 0.0,* `LAST(INTEGER)` *when* `x` *is plus or minus infinity, and zero when* `x` *is NaN.*

```
PROCEDURE NextAfter(x, y: T): T RAISES {FloatMode.Trap};
```

Return the next representable neighbor of `x` *in the direction towards* `y`. *If* `x = y`, *return* `x`.

```
PROCEDURE CopySign(x, y: T): T;
```

Return `x` *with the sign of* `y`.

```
PROCEDURE Finite(x: T): BOOLEAN;
```

Return `TRUE` if `x` is strictly between minus infinity and plus infinity. This always returns `TRUE` on non-IEEE implementations.

```
PROCEDURE IsNaN(x: T): BOOLEAN;
```

Return `FALSE` if `x` represents a numerical (possibly infinite) value, and `TRUE` if `x` does not represent a numerical value. For example, on IEEE implementations, returns `TRUE` if x is a NaN, `FALSE` otherwise.

```
PROCEDURE Sign(x: T): [0..1];
```

Return the sign bit `x`. For non-IEEE implementations, this is the same as `ORD(x >= 0)`; for IEEE implementations, `Sign(-0) = 1` and `Sign(+0) = 0`.

```
PROCEDURE Differs(x, y: T): BOOLEAN;
```

Return `(x < y OR y < x)`. Thus, for IEEE implementations, `Differs(NaN,x)` is always `FALSE`; for non-IEEE implementations, `Differs(x,y)` is the same as `x # y`.

```
PROCEDURE Unordered(x, y: T): BOOLEAN;
```

Return `NOT (x <= y OR y <= x)`. Thus, for IEEE implementations, `Unordered(NaN, x)` is always `TRUE`; for non-IEEE implementations, `Unordered(x, y)` is always `FALSE`.

```
PROCEDURE Sqrt(x: T): T RAISES {FloatMode.Trap};
```

Return the square root of `T`. This must be correctly rounded if `FloatMode.IEEE` is `TRUE`.

```
TYPE IEEEClass =
  {SignalingNaN, QuietNaN, Infinity, Normal, Denormal, Zero};
```

```
PROCEDURE Class(x: T): IEEEClass;
```

Return the IEEE number class containing `x`. On non-IEEE systems, the result will be `Normal` or `Zero`.

```
PROCEDURE FromDecimal(
    sign: [0..1];
    READONLY digits: ARRAY OF [0..9];
    exp: INTEGER): T RAISES {FloatMode.Trap};
```

Convert from floating-decimal to type `T`.

Let `F` denote the nonnegative, floating-decimal number

```
digits[0] . digits[1] ... digits[LAST(digits)] * 10^exp
= sum(i, digits[i] * 10^(exp - i))
```

The result of `FromDecimal` is the number `(-1)^sign * F`, rounded to a value of type `T`.

The procedure `FromDecimal` is a floating-point operation, just like `+` and `*`, in the sense that it rounds its ideal result correctly, observing the current rounding mode, and it sets flags and raises traps by the usual rules. On IEEE implementations, it returns minus zero when `F` is sufficiently small and `sign=1`.

```
TYPE DecimalApprox = RECORD
    class: IEEEClass;
    sign: [0..1];
    len: [1..R.MaxSignifDigits];
    digits: ARRAY[0..R.MaxSignifDigits-1] OF [0..9];
    exp: INTEGER;
    errorSign: [-1..1]
  END;

PROCEDURE ToDecimal(x: T): DecimalApprox;
```

Convert from type `T` to floating-decimal.

Let `D` denote `ToDecimal(x)`. Then, `D.class = Class(x)` and `D.sign = Sign(x)`. The other fields are defined only when `D.class` is either `Normal` or `Denormal`. In those cases, the values `D.len`, `D.digits[0]` through `D.digits[D.len-1]`, and `D.exp` encode a floating-decimal number `F` with the property that `(-1)^D.sign * F` approximates `x` in a sense discussed below. The encoding is such that

```
F = digits[0] . digits[1] ... digits[len - 1]  *  10^exp
  = sum(i, digits[i] * 10^(exp - i))
```

and

```
ABS(x) = F * (1 + errorSign * epsilon)
```

where `epsilon` is small and positive. In particular, `D.errorSign` is `+1`, `0`, or `-1` according as `ABS(x)` is larger than, equal to, or smaller than `F`.

The current rounding mode determines the sense in which the floating-decimal number `(-1)^sign * F` approximates `x`, but in a slightly subtle way. Define the opposite of a directed rounding mode by reversing the direction, as follows:

```
 Opp(TowardPlusInfinity) := TowardMinusInfinity
Opp(TowardMinusInfinity) := TowardPlusInfinity
        Opp(TowardZero) := AwayFromZero
```

Note that `AwayFromZero` isn't actually a rounding mode, but it is clear what it would mean if it were. For all other rounding modes `M`, we define `Opp(M) = M`. If the current rounding mode is `M`, the call `ToDecimal(x)` returns a floating-decimal number that `FromDecimal` would convert, under rounding mode `Opp(M)`, back to `x`. Among all such numbers, the returned value has as few digits as possible. This implies that both `D.digits[0]` and `D.digits[D.len-1]` are nonzero. If there is a tie for having the fewest digits, the tying number closest to `x` wins. If there is also a tie for being closest to `x`, it must be a two-way tie and the number whose last digit is even wins.

Unlike `FromDecimal`, `ToDecimal` never sets a `FloatMode.Flag` and never raises `FloatMode.Trap`.

The idea of converting to decimal by retaining just as many digits as are necessary to convert back to binary exactly was popularized by Guy L. Steele Jr. and Jon L White [17]. David M. Gay pointed out the importance, in this context, of demanding that the conversion to binary handle mid-point cases by a known rule [6]. For example, in IEEE double precision, the floating-decimal number `1e23` is precisely halfway between two adjacent floating-binary numbers. If conversion to binary were allowed to go either way in such a mid-point case, conversion to decimal would have to avoid producing the simple number `1e23`, producing instead either `1.0000000000000001e23` or `9.999999999999999e22`. We believe the idea of combining the Steele/White style of automatic precision control with directed rounding by using opposite rounding modes, as above, is new with Lyle Ramshaw.

```
END Float.
```

2.6 FloatMode

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

```
INTERFACE FloatMode;

CONST IEEE: BOOLEAN = ...;
```

`TRUE` for fully compliant IEEE implementations.

```
EXCEPTION Failure;
```

Raised by attempts to set modes that are not supported by the implementation.

```
TYPE RoundingMode =
  {NearestElseEven, TowardMinusInfinity, TowardPlusInfinity,
```

```
  TowardZero, NearestElseAwayFromZero, IBM370, Other};
```

Rounding modes. The first four are the IEEE modes. A VAX always does `NearestElseAwayFromZero`.

```
CONST RoundDefault: RoundingMode = ...;
```

Implementation-dependent: the default mode for rounding arithmetic operations, used by a newly forked thread. This also specifies the behavior of the `ROUND` operation in half-way cases.

```
PROCEDURE SetRounding(md: RoundingMode) RAISES {Failure};
```

Change the rounding mode for the calling thread to `md`, or raise the exception if this cannot be done. This affects the implicit rounding in floating-point operations; it does not affect the `ROUND` operation. Generally this can be done only on IEEE implementations and only if `md` is an IEEE mode.

```
PROCEDURE GetRounding(): RoundingMode;
```

Return the rounding mode for the calling thread.

```
TYPE Flag = {Invalid, Inexact, Overflow, Underflow,
  DivByZero, IntOverflow, IntDivByZero};
```

Associated with each thread is a set of boolean status flags recording whether the condition represented by the flag has occurred in the thread since the flag was last reset. The meaning of the first five flags is defined precisely in the IEEE floating point standard; roughly they mean:

`Invalid` = invalid argument to an operation.

`Inexact` = an operation produced an inexact result.

`Overflow` = a floating-point operation produced a result whose absolute value is too large to be represented.

`Underflow` = a floating-point operation produced a result whose absolute value is too small to be represented.

`DivByZero` = floating-point division by zero.

The meaning of the last two flags is:

`IntOverflow` = an integer operation produced a result outside the representable range.

`IntDivByZero` = integer `DIV` or `MOD` by zero.

```
CONST NoFlags = SET OF Flag {};
```

```
PROCEDURE GetFlags(): SET OF Flag;
```

Return the set of flags for the current thread.

```
PROCEDURE SetFlags(s: SET OF Flag)
  : SET OF Flag RAISES {Failure};
```

Set the flags for the current thread to `s`*, and return their previous values.*

```
PROCEDURE ClearFlag(f: Flag);
```

Turn off the flag `f` *for the current thread.*

```
EXCEPTION Trap(Flag);
```

```
TYPE Behavior = {Trap, SetFlag, Ignore};
```

The behavior of an operation that causes one of the flag conditions is either:

`Ignore` = return some result and do nothing.

`SetFlag` = return some result and set the condition flag. For IEEE implementations, the result of the operation is defined by the standard.

`Trap` = possibly set the condition flag; in any case raise the `Trap` exception with the appropriate flag as the argument.

```
PROCEDURE SetBehavior(f: Flag; b: Behavior) RAISES {Failure};
```

Set the behavior of the current thread for the flag `f` *to be* `b`*, or raise* `Failure` *if this cannot be done.*

```
PROCEDURE GetBehavior(f: Flag): Behavior;
```

Return the behavior of the current thread for the flag `f`*.*

```
END FloatMode.
```

2.7 Lex

The `Lex` interface provides procedures for reading strings, booleans, integers, and floating-point numbers from an input stream.

```
INTERFACE Lex;
```

```
IMPORT FloatMode, Rd, Word;
```

```
EXCEPTION Error;
```

```
CONST
   Blanks = SET OF CHAR{
     ' ', '\t', '\n', '\r', '\013' (* vertical tab *), '\f'};
   NonBlanks = SET OF CHAR{'!' .. '~'};
```

Each of the procedures in this interface reads a specified prefix of the characters in the reader passed to the procedure, and leaves the reader positioned immediately after that prefix, perhaps at end-of-file. Each procedure may call `Rd.UngetChar` after its final call on `Rd.GetChar`.

```
PROCEDURE Scan(
    rd: Rd.T; READONLY cs: SET OF CHAR := NonBlanks): TEXT
  RAISES {Rd.Failure};
```

Read the longest prefix of `rd` composed of characters in `cs` and return that prefix as a `TEXT`.

```
PROCEDURE Skip(
    rd: Rd.T; READONLY cs: SET OF CHAR := Blanks)
  RAISES {Rd.Failure};
```

Read the longest prefix of `rd` composed of characters in `cs` and discard it.

Whenever a specification of one of the procedures mentions skipping blanks, this is equivalent to performing the call `Skip(rd, Blanks)`.

```
PROCEDURE Match(rd: Rd.T; t: TEXT)
  RAISES {Rd.Failure, Error};
```

Read the longest prefix of `rd` that is also a prefix of `t`. Raise `Error` if that prefix is not, in fact, equal to all of `t`.

```
PROCEDURE Bool(rd: Rd.T): BOOLEAN RAISES {Rd.Failure, Error};
```

Read a boolean from `rd` and return its value.

`Bool` skips blanks, then reads the longest prefix of `rd` that is a prefix of a `Boolean` in the following grammar:

```
Boolean = "F" "A" "L" "S" "E" | "T" "R" "U" "E".
```

The case of letters in a `Boolean` is not significant. If the prefix read from `rd` is an entire `Boolean`, `Bool` returns that boolean; else it raises `Error`.

```
PROCEDURE Int(rd: Rd.T; defaultBase: [2..16] := 10)
  : INTEGER RAISES {Rd.Failure, Error, FloatMode.Trap};
PROCEDURE Unsigned(rd: Rd.T; defaultBase: [2..16] := 16)
  : Word.T RAISES {Rd.Failure, Error, FloatMode.Trap};
```

Read a number from `rd` and return its value.

Each procedure skips blanks, then reads the longest prefix of `rd` that is a prefix of a `Number` as defined by the grammar below. If `defaultBase` exceeds 10, then the procedure scans for a `BigBaseNum`; otherwise it scans for a `SmallBaseNum`. The effect of this rule is that the letters `'a'` through `'f'` and `'A'` through `'F'` stop the scan unless either the `defaultBase` or the explicitly provided base exceeds 10. `Unsigned` omits the scan for a `Sign`.

```
Number       = [Sign] (SmallBaseNum | BigBaseNum).
SmallBaseNum = DecVal | BasedInt.
BigBaseNum   = HexVal | BasedInt.
BasedInt     = SmallBase "_" DecVal | BigBase "_" HexVal.
DecVal       = Digit {Digit}.
HexVal       = HexDigit {HexDigit}.
Sign         = "+" | "-".
SmallBase    = "2" | "3" | ... | "10".
BigBase      = "11" | "12" | ... | "16".
Digit        = "0" | "1" | ... | "9".
HexDigit     = Digit | "A" | "B" | "C" | "D" | "E" | "F"
                     | "a" | "b" | "c" | "d" | "e" | "f".
```

If the prefix read from `rd` is an entire `Number` (as described above), the corresponding number is returned; else `Error` is raised.

If an explicit base is given with an underscore, it is interpreted in decimal. In this case, the digits in `DecVal` or `HexVal` are interpreted in the explicit base, else they are interpreted in the `defaultBase`.

Both procedures may raise `FloatMode.Trap(IntOverflow)`. They raise `Error` if some digit in the value part is not a legal digit in the chosen base.

```
PROCEDURE Real(rd: Rd.T): REAL
  RAISES {Rd.Failure, Error, FloatMode.Trap};
PROCEDURE LongReal(rd: Rd.T): LONGREAL
  RAISES {Rd.Failure, Error, FloatMode.Trap};
PROCEDURE Extended(rd: Rd.T): EXTENDED
  RAISES {Rd.Failure, Error, FloatMode.Trap};
```

Read a real number from `rd` *and return its value.*

Each procedure skips blanks, then reads the longest prefix of `rd` that is a prefix of a floating-decimal number `Float` in the grammar:

```
Float  = [Sign] FloVal [Exp].
FloVal = {Digit} (Digit | Digit "." | "." Digit) {Digit}.
Exp    = Marker [Sign] Digit {Digit}.
Marker = ("E" | "e" | "D" | "d" | "X" | "x").
```

where `Sign` and `Digit` are as defined above. If the prefix read from `rd` is an entire `Float`, that `Float` is converted to a `REAL`, `LONGREAL`, or `EXTENDED` using the routine `FromDecimal` in the appropriate instance of the `Float` generic interface; else `Error` is raised. Note that the exponent of `Float` can be introduced with any of the six characters `'e'`, `'E'`, `'d'`, `'D'`, `'x'`, or `'X'`, independent of the target type of the conversion.

On IEEE implementations, the syntax for `Float` is extended as follows:

```
Float   = [Sign] FloVal [Exp] | [Sign] IEEEVal.
IEEEVal = "I" "N" "F" "I" "N" "I" "T" "Y" | "I" "N" F"
        | "N" "A" "N".
```

The case of letters in an `IEEEVal` is not significant. The `FloatMode.Trap` exception may be raised with any of the arguments `Overflow`, `Underflow`, or `Inexact`.

```
END Lex.
```

2.8 Fmt

The `Fmt` interface provides procedures for formatting numbers and other data as text.

```
INTERFACE Fmt;

IMPORT Word, Real AS R, LongReal AS LR, Extended AS ER;

PROCEDURE Bool(b: BOOLEAN): TEXT;
```

Format `b` *as* "TRUE" *or* "FALSE".

```
PROCEDURE Char(c: CHAR): TEXT;
```

Return a text containing the character `c`.

```
TYPE Base = [2..16];

PROCEDURE Int(n: INTEGER; base: Base := 10): TEXT;
PROCEDURE Unsigned(n: Word.T; base: Base := 16): TEXT;
```

Format the signed or unsigned number `n` *in the specified base.*

The value returned by `Int` or `Unsigned` never contains upper-case letters, and it never starts with an explicit base and underscore. For example, to render an unsigned number `N` in hexadecimal as a legal Modula-3 literal, you must write something like:

```
"16_" & Fmt.Unsigned(N, 16)
```

```
TYPE Style = {Sci, Fix, Auto};

PROCEDURE Real(
```

```
    x: REAL;
    style := Style.Auto;
    prec: CARDINAL := R.MaxSignifDigits - 1;
    literal := FALSE)
  : TEXT;
PROCEDURE LongReal(
    x: LONGREAL;
    style := Style.Auto;
    prec: CARDINAL := LR.MaxSignifDigits - 1;
    literal := FALSE)
  : TEXT;
PROCEDURE Extended(
    x: EXTENDED;
    style := Style.Auto;
    prec: CARDINAL := ER.MaxSignifDigits - 1;
    literal := FALSE)
  : TEXT;
```

Format the floating-point number `x`*.*

Overview. `Style.Sci` gives scientific notation with fields padded to fixed widths, suitable for making a table. The parameter `prec` specifies the number of digits after the decimal point—that is, the relative precision.

`Style.Fix` gives fixed point, with `prec` once again specifying the number of digits after the decimal point—in this case, the absolute precision. The results of `Style.Fix` have varying widths, but they will form a table if they are right-aligned (using `Fmt.Pad`) in a sufficiently wide field.

`Style.Auto` is not intended for tables. It gives scientific notation with at most `prec` digits after the decimal point for numbers that are very big or very small. There may be fewer than `prec` digits after the decimal point because trailing zeros are suppressed. For numbers that are neither too big nor too small, it formats the same significant digits—at most `prec+1` of them—in fixed point, for greater legibility.

All styles omit the decimal point unless it is followed by at least one digit.

Setting `literal` to `TRUE` alters all styles as necessary to make the result a legal Modula-3 literal of the appropriate type.

Accuracy. As discussed in the `Float` interface, the call `ToDecimal(x)` converts `x` to a floating-decimal number with automatic precision control [17, 6]: Just enough digits are retained to distinguish `x` from other values of type `T`, which implies that at most `T.MaxSignifDigits` are retained. The `Real`, `LongReal`, and `Extended` procedures format those digits as an appropriate string of characters. If the precision requested by `prec` is higher than the automatic precision provided by `ToDecimal(x)`, they append trailing zeros. If the precision requested by `prec` is lower, they round `ToDecimal(x)` as necessary, obeying the current rounding mode. Because they exploit the `errorSign` field of the record `ToDecimal(x)` in doing this rounding, they get the same result that rounding `x` itself would give.

As a consequence, setting `prec` higher than `T.MaxSignifDigits-1` in `Style.Sci` isn't very useful: The trailing digits of all of the resulting numbers will be zero. Setting `prec` higher than `T.MaxSignifDigits-1` in `Style.Auto` actually has no effect at all, since trailing zeros are suppressed.

Details. We restrict ourselves at first to those cases where `Class(x)` is either `Normal` or `Denormal`.

In those cases, `Style.Sci` returns: a minus sign or blank, the leading nonzero digit of `x`, a decimal point, `prec` more digits of `x`, a character `'e'`, a minus sign or plus sign, and `T.MaxExpDigits` of exponent (with leading zeros as necessary). When `prec` is zero, the decimal point is omitted.

`Style.Fix` returns: a minus sign if necessary, one or more digits, a decimal point, and `prec` more digits—never any blanks. When `prec` is zero, the decimal point is omitted.

`Style.Auto` first formats `x` as in `Style.Sci`, using scientific notation with `prec` digits after the decimal point. Call this intermediate result `R`.

If the exponent of `R` is at least `6` in magnitude, `Style.Auto` leaves `R` in scientific notation, but condenses it by omitting all blanks, plus signs, trailing zero digits, and leading zeros in the exponent. If this leaves no digits after the decimal point, the decimal point itself is omitted.

If the exponent of `R` is at most `5` in magnitude, `Style.Auto` reformats the digits of `R` in fixed point, first deleting any trailing zeros and then adding leading or trailing zeros as necessary to bridge the gap from the digits of `R` to the unit's place.

For example:

```
Fmt.Real(1.287e6,  Style.Auto, prec := 2) = "1.29e6"
Fmt.Real(1.297e6,  Style.Auto, prec := 2) = "1.3e6"
Fmt.Real(1.297e5,  Style.Auto, prec := 2) = "130000"
Fmt.Real(1.297e-5, Style.Auto, prec := 2) = "0.000013"
Fmt.Real(1.297e-6, Style.Auto, prec := 2) = "1.3e-6"
Fmt.Real(9.997e5,  Style.Auto, prec := 2) = "1e6"
Fmt.Real(9.997e-6, Style.Auto, prec := 2) = "0.00001"
```

`Style.Sci` handles zero by replacing the entire exponent field by blanks, for example: `"␣0.00␣␣␣␣"`. `Style.Fix` renders zero with all digits zero; for example, `"0.00"`. `Style.Auto` renders zero as `"0"`. On IEEE implementations, the value minus zero is rendered as a negative number.

Also on IEEE implementations, `Style.Sci` formats infinities or NaN's with a minus sign or blank, the string `"Infinity"` or `"NaN"`, and enough trailing blanks to get the correct overall width. `Style.Fix` and `Style.Auto` omit the blanks. In `Style.Sci`, if `"Infinity"` doesn't fit, `"Inf"` is used instead.

Setting `literal` to `TRUE` alters things as follows: Numbers that are rendered without a decimal point when `literal` is `FALSE` have a decimal point and one trailing zero appended to their digits. For the routines `Fmt.LongReal` and `Fmt.Extended`, an exponent field of `d0` or `x0` is appended to numbers in fixed point and `'d'` or `'x'` is used, rather than `'e'`, to introduce the exponents of numbers in scientific notation. On IEEE implementations, the string `"Infinity"` is replaced by `"1.0/0.0"`, `"1.0d0/0.0d0"`, or `"1.0x0/0.0x0"` as appropriate, and `"NaN"` is similarly replaced by a representation of the quotient `0/0`. (Unfortunately, these quotient strings are so long that they may ruin the formatting of `Style.Sci` tables when `prec` is small and `literal` is `TRUE`.)

```
TYPE Align = {Left, Right};
```

```
PROCEDURE Pad(
    text: TEXT;
    length: CARDINAL;
    padChar: CHAR := ' ';
    align: Align := Align.Right): TEXT;
```

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`.

```
PROCEDURE F(fmt: TEXT; t1, t2, t3, t4, t5: TEXT := NIL)
  : TEXT;
```

Uses `fmt` as a format string. The result is a copy of `fmt` in which all format specifiers have been replaced, in order, by the text arguments `t1`, `t2`, etc.

A format specifier contains a field width, alignment and one of two padding characters. The procedure `F` evaluates the specifier and replaces it by the corresponding text argument padded as it would be by a call to `Pad` with the specified field width, padding character and alignment.

The syntax of a format specifier is:

```
%[-]{0-9}s
```

that is, a percent character followed by an optional minus sign, an optional number and a compulsory terminating `s`.

If the minus sign is present the alignment is `Align.Left`, otherwise it is `Align.Right`. The alignment corresponds to the `align` argument to `Pad`.

The number specifies the field width (this corresponds to the `length` argument to `Pad`). If the number is omitted it defaults to zero.

If the number is present and starts with the digit `0` the padding character is `'0'`; otherwise it is the space character. The padding character corresponds to the `padChar` argument to `Pad`.

It is a checked runtime error if `fmt` is `NIL` or the number of format specifiers in `fmt` is not equal to the number of non-nil arguments to `F`.

Non-nil arguments to `F` must precede any `NIL` arguments; it is a checked runtime error if they do not.

If `t1` to `t5` are all `NIL` and `fmt` contains no format specifiers, the result is `fmt`.

Examples:

```
F("%s %s\n", "Hello", "World") returns "Hello World\n".
F("%s", Int(3))                returns "3"
F("%2s", Int(3))               returns " 3"
F("%-2s", Int(3))              returns "3 "
F("%02s", Int(3))              returns "03"
F("%-02s", Int(3))             returns "30"
F("%s", "%s")                  returns "%s"
F("%s% tax", Int(3))           returns "3% tax"
```

The following examples are legal but pointless:

```
F("%-s", Int(3))               returns "3"
F("%0s", Int(3))               returns "3"
F("%-0s", Int(3))              returns "3"
```

```
PROCEDURE FN(fmt: TEXT; READONLY texts: ARRAY OF TEXT)
  : TEXT;
```

Similar to `F` but accepts an array of text arguments. It is a checked runtime error if the number of format specifiers in `fmt` is not equal to `NUMBER(texts)` or if any element of `texts` is `NIL`. If `NUMBER(texts) = 0` and `fmt` contains no format specifiers the result is `fmt`.

Example:

```
FN("%s %s %s %s %s %s %s",
  ARRAY OF TEXT{"Too", "many", "arguments",
    "for", "F", "to", "handle"})
```

returns `"Too many arguments for F to handle"`.

```
END Fmt.
```

3 Data Structures

3.1 Sequence

`Sequence` is a generic interface defining extensible sequences. Elements can be added or removed at either end of a sequence; they can also be accessed or updated at specified indexes. The expected cost of every method of a sequence is constant.

```
GENERIC INTERFACE Sequence(Elem);
```

Where `Elem.T` is a type that is not an open array type.

```
TYPE
  T <: Public;
  Public = OBJECT METHODS
    init(sizeHint: CARDINAL := 5): T;
    fromArray(READONLY a: ARRAY OF Elem.T): T;
    addhi(READONLY x: Elem.T);
    addlo(READONLY x: Elem.T);
    remhi(): Elem.T;
    remlo(): Elem.T;
    put(i: CARDINAL; READONLY x: Elem.T);
    size(): CARDINAL;
    gethi(): Elem.T;
    getlo(): Elem.T;
    get(i: CARDINAL): Elem.T
  END;
```

A `Sequence(Elem).T` (or just a *sequence*) represents an extensible sequence of `Elem.T`s.

The first group of methods have side effects on the sequence. The call

```
s.init(sizeHint)
```

initializes `s` to be the empty sequence. Furthermore `init` assumes that at least `sizeHint` elements will be added to the sequence; these operations may be executed more efficiently than if `sizeHint` was defaulted. The call

```
s.fromArray(a)
```

initializes `s` to be the sequence with elements `a[0], ..., a[LAST(a)]`. The call

```
s.addhi(x)
```

appends `x` to the end of `s`. Thus it does not change the index of any existing element. The call

```
s.addlo(x)
```

appends `x` to the front of `s`. Thus it increases the index of all existing elements by one. The call

```
s.remhi()
```

removes and returns the last element of `s`. Thus it does not change the index of any of `s`'s other elements. If `s` is empty, `s.remhi()` causes a checked runtime error. The call

```
s.remlo()
```

removes and returns the first element of `s`. Thus it decreases the index of all other elements of `s` by one. If `s` is empty, `s.remlo()` causes a checked runtime error. The call

```
s.put(i, x)
```

replaces element `i` of `s` with `x`. Element `0` is the first element. It is a checked runtime error unless `i` is less than `s.size()`.

The second group of methods have no side effect on the sequence. The call

```
s.size()
```

returns the number of elements in `s`. The call

```
s.get(i)
```

returns element `i` of `s`. It is a checked runtime error unless `i` is less than `s.size()`. The call

```
s.gethi()
```

returns the last element of `s`; that is, it is equivalent to `s.get(s.size()-1)`. The call

```
s.getlo()
```

returns the first element of `s`; that is, it is equivalent to `s.get(0)`.

```
PROCEDURE Cat(s, t: T): T;
```

Return a sequence whose elements are the concatenation of `s` *and* `t`*.*

```
PROCEDURE Sub(s: T; start: CARDINAL;
    length: CARDINAL := LAST(CARDINAL)): T;
```

Return a sub-sequence of `s`*: empty if* `start >= t.size()` *or* `length = 0`*; otherwise the subsequence ranging from* `start` *to the minimum of* `start+length-1` *and* `s.size()-1`*.*

`Cat` and `Sub` create new sequences; they have no side-effects.

Sequences are unmonitored: a client accessing a sequence from multiple threads must ensure that if two operations are active concurrently, then neither of them has side effects on the sequence.

```
END Sequence.
```

The standard instances are named `AtomSeq`, `IntSeq`, `RefSeq`, and `TextSeq`.

3.2 Atom

An `Atom.T` is a unique representative for a set of equal texts (like a Lisp atomic symbol)

```
INTERFACE Atom;

TYPE T <: REFANY;

PROCEDURE FromText(t: TEXT): T;
```

Return the unique atom `a` *such that for any text* `u`*, if* `Text.Equal(u, t)`*, then* `FromText(u) = a`*.*

```
PROCEDURE ToText(a: T): TEXT;
```

Return a text `t` *such that* `FromText(t) = a`*.*

```
PROCEDURE Equal(a1, a2: T): BOOLEAN;
```

Return `a1 = a2`*.*

```
PROCEDURE Hash(a: T): INTEGER;
```

Return a hash code for `a` *by taking the image of* `ToText(a)` *under some fixed hash function.*

```
PROCEDURE Compare(a1, a2: T): [-1..1];
```

Cause a checked runtime error.

```
END Atom.
```

`Compare` causes a checked runtime error because there is no default order on atoms.

3.3 List and ListSort

The generic interface `List` provides operations on linked lists of arbitrary element types.

```
GENERIC INTERFACE List(Elem);
```

Where `Elem.T` is not an open array type and `Elem` contains

```
PROCEDURE Equal(k1, k2: Elem.T): BOOLEAN;
```

`Equal` may be declared with a parameter mode of either `VALUE` or `READONLY`, but not `VAR`.

```
TYPE T = OBJECT head: Elem.T; tail: T END;
```

A `List.T` represents a linked list of items of type `Elem.T`.

None of the operations of this interface modify the `head` field of an existing list element. Operations that may modify the `tail` field of existing list elements are called *destructive*. By convention, their names end in `D`.

```
PROCEDURE Cons(READONLY head: Elem.T; tail: T): T;
```

Equivalent to `NEW(T, head := head, tail := tail)`.

```
PROCEDURE List1(READONLY e1: Elem.T): T;
```

Return a list containing the single element `e1`.

```
PROCEDURE List2(READONLY e1, e2: Elem.T): T;
```

Return a list containing the element sequence `e1`, `e2`.

```
PROCEDURE List3(READONLY e1, e2, e3: Elem.T): T;
```

Return a list containing the element sequence `e1`, `e2`, `e3`.

```
PROCEDURE FromArray(READONLY e: ARRAY OF Elem.T): T;
```

Return a list containing the elements of `e` in order.

```
PROCEDURE Length(l: T): CARDINAL;
```

Return the number of elements of `l`.

```
PROCEDURE Nth(l: T; n: CARDINAL): Elem.T;
```

Return element `n` of list `l`. Element 0 is `l.head`, element 1 is `l.tail.head`, etc. Cause a checked runtime error if `n >= Length(l)`.

```
PROCEDURE Member(l: T; READONLY e: Elem.T): BOOLEAN;
```

Return `TRUE` if some element of `l` is equal to `e`, else return `FALSE`. The comparison is performed by `Elem.Equal`.

```
PROCEDURE Append(l1: T; l2: T): T;
PROCEDURE AppendD(l1: T; l2: T): T;
```

Append two lists together, returning the new list. `Append` does this by making a copy of the cells of `l1`; `AppendD` modifies the `tail` field in the last cell of `l1`.

```
PROCEDURE Reverse(l: T): T;
PROCEDURE ReverseD(l: T): T;
```

Return a list containing the elements of `l` in reverse order. `Reverse` copies the cells; `ReverseD` modifies the `tail` fields of the existing cells.

```
END List.
```

The standard instances are named `AtomList`, `IntList`, `RefList`, and `TextList`.

The generic interface `ListSort` extends the generic interface `List` with sorting operations.

```
GENERIC INTERFACE ListSort(Elem, ElemList);
```

Where `Elem.T` is not an open array type, `Elem` contains

```
PROCEDURE Compare(e1, e2: Elem.T): [-1..1];
```

and `ElemList` equals `List(Elem)`. `Compare` must be a total order. It may be declared with any parameter mode, but must have no visible side-effects.

```
TYPE T = ElemList.T;

PROCEDURE Sort(l: T; c := Elem.Compare): T;
PROCEDURE SortD(l: T; c := Elem.Compare): T;
```

Sort a list in ascending order using `c` to compare pairs of elements of `l`.

The implementation is time- and cons-efficient but not guaranteed to be stable. `Sort` copies the cells; `SortD` modifies the `tail` fields of the existing cells.

```
END ListSort.
```

The standard instances are named `AtomListSort`, `IntListSort`, `RefListSort`, and `TextListSort`. `AtomListSort` and `RefListSort` are useful only if you supply a non-default comparison procedure.

3.4 Sx

An `Sx.T` is a symbolic expression represented as a recursive linked list structure, as in Lisp. This interface provides routines for reading and printing symbolic expressions, as well as some convenience procedures for manipulating them. The syntax of an `Sx.T` is as follows:

```
Sx = Char | Text | Int | Real | Longreal | Extended
   | Atom | Boolean | "(" List ")".

List =  {Sx}.
```

A `Char` is a Modula-3 character literal; the corresponding `Sx.T` is of type `REF CHAR`.

A `Text` is a Modula-3 text literal. The corresponding `Sx.T` is a `TEXT`.

An `Int` is a Modula-3 integer literal, possibly preceded by a plus sign (`+`) or minus sign (`-`). The corresponding `Sx.T` is of type `REF INTEGER`.

A `Real`, `Longreal`, or `Extended` is a floating-decimal number parsed using the grammar for `Float` specified in the `Lex` interface. The corresponding `Sx.T` is of type `REF REAL`, `REF LONGREAL` or `REF EXTENDED`, depending on whether the letter introducing the exponent is `'e'`, `'d'`, or `'x'`. If there is no exponent, the result will be of type `REF REAL`.

An `Atom` is either (1) a Modula-3 identifier, or (2) a non-empty sequence of characters from the set

```
! # $ % & * + - . / : < = > ? @ [ ] ^ _ { } ~
```

or (3) a sequence of characters and escape sequences surrounded by vertical bars (`|`s). The escape sequences are the same as those allowed in Modula-3 text literals, with the addition of `\|` to allow an atom to contain `|`. In all three cases, the corresponding `Sx.T` is an `Atom.T`.

For example, the following are valid atoms:

```
A1
+=
|1\||
```

A `Boolean` is either `TRUE` or `FALSE`; the corresponding `Sx.T` is of type `Atom.T`; in other words, this is not a distinct type.

The `Sx.T` corresponding to a `List` is a `RefList.T` containing the items of the list in order.

The tokens of an `Sx.T` can be separated by arbitrary sequences of blanks, tabs, newlines, carriage returns, form feeds, and vertical tabs, which are ignored. (These are the same whitespace characters that are ignored between tokens of a Modula-3 program.) They can also be separated by comments, which begin with a semicolon and end with newline.

The syntax of tokens can be extended with the `SetReadMacro` procedure.

```
INTERFACE Sx;

IMPORT Atom, Rd, RefList, Thread, Wr;

TYPE T = REFANY;

EXCEPTION
  ReadError(TEXT);
  PrintError(TEXT);

PROCEDURE FromChar(c: CHAR): REF CHAR;
```

Return a `Char` with value `c`.

```
PROCEDURE FromInt(i: INTEGER): REF INTEGER;
```

Return an `Int` with value `i`.

```
PROCEDURE FromReal(r: REAL): REF REAL;
```

Return a `Real` with value `r`.

```
PROCEDURE FromLongReal(r: LONGREAL): REF LONGREAL;
```

Return a `Longreal` with value `r`.

```
PROCEDURE FromExtended(r: EXTENDED): REF EXTENDED;
```

Return an `Extended` with value `r`.

```
PROCEDURE FromBool(b: BOOLEAN): Atom.T;
```

Return a `Boolean`. If `b` is `TRUE`, return `Sx.True`. Otherwise, return `Sx.False`.

The `From...` procedures do not necessarily perform an allocation: if the same value is passed to two calls, the same reference may be returned. As a consequence, clients should not modify the referent of a reference returned by any of these procedures.

Each `REF CHAR`, `REF INTEGER`, `REF REAL`, `REF LONGREAL`, `REF EXTENDED`, `TEXT`, or `Atom.T`, no matter how constructed, is an `Sx.T`.

```
VAR (*CONST*) True, False: Atom.T;
```

`True = Atom.FromText("TRUE")`, `False = Atom.FromText("FALSE")`.

```
PROCEDURE Read(rd: Rd.T; syntax: Syntax := NIL): T
  RAISES {ReadError, Rd.EndOfFile, Thread.Alerted};
```

Read and return a symbolic expression from `rd`, ignoring whitespace and comments. If `syntax` is `NIL`, use the syntax described above; otherwise use any read macros that have been registered in `syntax`.

```
PROCEDURE ReadDelimitedList(
    rd: Rd.T; delim : CHAR; syntax: Syntax := NIL): RefList.T
  RAISES {ReadError, Thread.Alerted};
```

Repeatedly read symbolic expressions from `rd`*, ignoring whitespace and comments, until the next character is* `delim`*; consume the delimiter and return the list of symbolic expressions that were read. Raise* `ReadError` *if there is a syntax error, including unexpected end of file.*

```
PROCEDURE Print(
    wr: Wr.T;
    sx: T;
    maxDepth: CARDINAL := LAST(CARDINAL);
    maxLength: CARDINAL := LAST(CARDINAL))
  RAISES {PrintError, Wr.Failure, Thread.Alerted};
```

Print the symbolic expression `sx` *on the writer* `wr`*, assuming the standard syntax.*

Each sublist will contain no more than `maxLength` elements; extra elements are replaced by an ellipsis (three dots). Any sublist nested at a depth greater than `maxDepth` is also replaced by an ellipsis. `Print` inserts `|` around atoms if necessary to ensure that they are readable. `Print` does not insert line-breaks or indentation to produce a human-readable ("pretty-printed") format for large symbolic expressions.

`Print` will raise `PrintError` if it tries to print something that is not "printable" (as defined below). If a list contains an unprintable element that is beyond the limits established by `maxDepth` and `maxLength`, `PrintError` may or may not be raised.

An object is said to be "printable" if it satisfies the following hypothetical predicate:

```
PROCEDURE Printable(x: REFANY): BOOLEAN =
  BEGIN
    TYPECASE x OF
    | NULL, REF CHAR, TEXT, REF INTEGER, REF REAL,
      REF LONGREAL, REF EXTENDED, Atom.T =>
        RETURN TRUE
    | RefList.T (list) => RETURN Printable(list.head) AND
                                 Printable(list.tail)
    ELSE
        RETURN FALSE
    END
 END Printable;
```

`Read(rd,NIL)` is guaranteed to return a printable value unless it raises an exception. Assuming the defaults for `syntax`, `maxDepth`, and `maxLength`, and assuming no exceptions are raised, `Read` and `Print` are "inverses".

```
TYPE Syntax <: REFANY;
```

A `Syntax` is a partial map from characters to read macros.

```
PROCEDURE CopySyntax(s: Syntax := NIL): Syntax;
```

Allocate and return a new syntax table whose contents are the same as `s` *or, if* `s = NIL`*, the same as the standard syntax table. The standard syntax table has no read macros.*

```
PROCEDURE SetReadMacro(s: Syntax; ch: CHAR; m: ReadMacro);
```

Set `s[ch] := m`*. It is a checked runtime error if* `s = NIL`*, if* `ch` *is a whitespace character, or if* `ch = ';'`*. It is allowed for* `m` *to be* `NIL`*; this has the effect of removing the mapping, if any, from* `ch` *to a readmacro.*

```
TYPE ReadMacro = OBJECT METHODS
    read(rd: Rd.T; s: Syntax): RefList.T
      RAISES {ReadError, Thread.Alerted}
  END;
```

If you pass a `Syntax s` to `Read` or `ReadDelimitedList`, then the reading algorithm is modified as follows. After skipping whitespace and comments, and before reading a token, the next character in the input stream is consumed and examined. If `s` defines a read macro for this character, then this read macro is called with the same arguments that were passed to `Read` or `ReadDelimitedList`. The resulting list is spliced into the current list being built. In particular, if the macro returns `NIL`, then everything it read is ignored; if the macro returns a single-element list, then that single element is inserted into the list being built. `ReadError` is raised if the macro returns a non-list or if it returns a multi-element list in a context where no list is being built, such as at the top level of `Read`.

For example, the following program fragment constructs a syntax table that extends the standard syntax in two ways. First, additional comments are supported by ignoring all characters between `{` and `}`. Second, an expression of the form `[e1 ... en]` is turned into the list `(ARRAY e1 ... en)`:

```
VAR syn := CopySyntax(); BEGIN
  SetReadMacro(syn, '{',
    NEW(ReadMacro, read := ReadComment));
  SetReadMacro(syn, '[',
    NEW(ReadMacro, read := ReadArray));
  ...
```

```
PROCEDURE ReadComment(
    self: ReadMacro; rd: Rd.T; <* UNUSED *> s: Syntax)
  : RefList.T =
  BEGIN
    WHILE NOT Rd.EOF() AND Rd.GetChar(rd) # '}' DO
      (* SKIP *)
    END;
    RETURN NIL
  END ReadComment;

VAR (*CONST*) arrayAtm := Atom.FromText("ARRAY");

PROCEDURE ReadArray(self: ReadMacro; rd: Rd.T; s: Syntax)
  : RefList.T =
  VAR elements := ReadDelimitedList(rd, ']', s);
  BEGIN
    RETURN RefList.List1(RefList.Cons(arrayAtm, elements))
  END ReadArray;
```

The call to `RefList.List1` in `ReadArray` is important. If it were omitted, then the text

```
(a b [c d])
```

would be read as

```
(a b ARRAY c d)
```

instead of the intended

```
(a b (ARRAY c d)).
```

```
END Sx.
```

3.5 Table

`Table` is a generic interface defining partial maps that support update and iteration.

```
GENERIC INTERFACE Table(Key, Value);
```

Where `Key.T` and `Value.T` are types that are not open array types and `Key` contains

```
PROCEDURE Equal(k1, k2: Key.T): BOOLEAN;
PROCEDURE Hash(k: Key.T): Word.T;
```

`Equal` must be an equivalence relation and `Hash` must respect that equivalence relation, in other words, if `Equal(k1, k2)`, then `Hash(k1)=Hash(k2)`.

`Hash` and `Equal` may be declared with a parameter mode of either `VALUE` or `READONLY`, but not `VAR`.

```
IMPORT Word;

TYPE
  T = OBJECT METHODS
    get(READONLY k: Key.T; VAR v: Value.T): BOOLEAN;
    put(READONLY k: Key.T; READONLY v: Value.T): BOOLEAN;
    delete(READONLY k: Key.T; VAR v: Value.T): BOOLEAN;
    size(): CARDINAL;
    iterate(): Iterator
  END;
  Iterator = OBJECT METHODS
    next(VAR k: Key.T; VAR v: Value.T): BOOLEAN
  END;
  Default <: T OBJECT METHODS
    init(sizeHint: CARDINAL := 0): Default;
    keyEqual(READONLY k1, k2: Key.T): BOOLEAN;
    keyHash(READONLY k: Key.T): Word.T
  END;
END Table.
```

A `Table(Key, Value).T`, or table, is a partial map from `Key.T`s to `Value.T`s. Actually, it turns out to be useful for a table to treat two different keys as if they are the same whenever they are equivalent according to some specified equivalence relation. For example, if you are creating a table with a `Key.T` of `TEXT`, you are likely to want `Text.Equal` as the equivalence relation.

Formally, a table `tbl` has the components:

`canon(tbl)`	a map on elements of `Key.T`
`map(tbl)`	a map from elements of `Key.T` to elements of `Value.T`

`canon(tbl)` represents an equivalence relation: `canon(tbl)(k)` is the *canonical representative* of all the keys that are equivalent to `k`. The domain of `map(tbl)` includes only canonical representatives, that is, elements in the range of `canon(tbl)`. The equivalence relation underlying `canon(tbl)` must be time-invariant. For example, it can't depend on the values of particular references since some garbage collectors move `REF` values.

The methods of an object `tbl` of type `Table.T` have the following specifications:

The call `tbl.get(k, v)` sets `v` to `map(tbl)(canon(tbl)(k))` and returns `TRUE` if `canon(tbl)(k)` is in `dom(map(tbl))`. Otherwise, it returns `FALSE` without changing `v`.

The call `tbl.put(k, v)` changes `map(tbl)(canon(tbl)(k))` to `v` and returns `TRUE` if `canon(k)` is in `dom(map(tbl))`. Otherwise, it sets the value of `map(tbl)(canon(tbl)(k))` to `v`, and returns `FALSE`.

The call `tbl.delete(k, v)` sets `v` to `map(tbl)(canon(tbl)(k))`, removes `(canon(tbl)(k), v)` from `map(tbl)`, and returns `TRUE` if `canon(tbl)(k)` is in `dom(map(tbl))`. Otherwise, it returns `FALSE` without changing `v`.

The call `tbl.size()` returns the size of `dom(map(tbl))`, that is, the number of entries in `tbl`.

The call `tbl.iterate()` returns an iterator, which is an object that can be used to iterate over the key-value pairs in `tbl`. See the definition of the type `Iterator` below.

If `i` is the result of the call `tbl.iterate()`, then the call `it.next(k, v)` selects an entry from `tbl` that has not already been returned by `i`, sets `k` and `v` to its key and value, and returns `TRUE`. If no entries remain, the call returns `FALSE` without setting `k` or `v`. It is a checked runtime error to call `next` after it has returned `FALSE`. The client must ensure that while an iterator is in use, the parent table is not modified.

The type `Default` is an implementation of `T` using chained hashing. The methods specific to an object `dflt` of type `Default` have the following specifications:

The call `dflt.init(sizeHint)` returns `dflt` after initializing it to a table with an empty `map(dflt)`. If `sizeHint` is greater than 0, `init` assumes that `put` will subsequently be called with at least `sizeHint` different keys; these calls on `put` may execute somewhat faster than if `sizeHint` was 0. The `init` method has side-effects on the table.

The call `dflt.keyEqual(k1, k2)` returns `Key.Equal(k1, k2)` and the call `dflt.keyHash(k)` returns `Key.Hash(k)`. The other methods call `keyEqual` and `keyHash` whenever they need to consult the table's equivalence relation. This means a subtype of `Default` can determine the equivalence relation by overriding `keyEqual` and `keyHash`, providing `keyEqual` implements an equivalence relation and `keyHash` respects that relation.

For efficiency, tables and their iterators are not monitored, so a client accessing a table from multiple threads must ensure that if two operations are active concurrently, then neither of them has side effects on the same table or iterator. The `T.put`, `T.delete`, and `Default.init` methods are the only ones with side effects on the table. An iterator's `next` method has side-effects on the iterator.

The standard instances are named `xyTbl`, for all combinations of `x` and `y` in the set `Atom, Int, Ref, Text`. The instances with `x = Ref` are useful only if you define a subtype overriding the `keyHash` and `keyEqual` methods.

3.6 SortedTable

`SortedTable` is a generic interface defining partial maps over a totally ordered domain.

```
GENERIC INTERFACE SortedTable(Key, Tbl);
```

Where `Key.T` is not an open array type, `Tbl` is a generic instance `Table(Key, Value)` (for some `Value` defining a type `T` that is not an open array type), and `Key` contains

```
PROCEDURE Compare(k1, k2: Key.T): [-1..1];
```

`Compare` must be a total order.

`Compare` may be declared with a parameter mode of either `VALUE` or `READONLY`, but not `VAR`.

```
TYPE
  T = Tbl.T OBJECT METHODS
    iterateOrdered(up: BOOLEAN := TRUE): Iterator
  END;
  Iterator = Tbl.Iterator OBJECT METHODS
    seek(READONLY key: Key.T)
  END;
  Default <: T OBJECT METHODS
    init(): Default;
    keyCompare(READONLY k1, k2: Key.T): [-1..1]
  END;
END SortedTable.
```

A `SortedTable(Key, Table(Key, Value)).T`, or sorted table, is a `Table(Key, Value).T` together with a total (linear) order on the keys of the table. Formally, a sorted table `tbl` has the additional component:

`le(tbl)` a total order on the values of `Key.T`

The total order `le(tbl)` must be time-invariant.

The methods have the following specifications:

The call `tbl.iterateOrdered(up)` returns an iterator, which is an object that can be used to iterate over all the key-value pairs in `tbl`, ordered by key. The order is increasing if `up` is `TRUE`, decreasing otherwise.

If `i` is the result of the call `tbl.iterateOrdered(up)`, then the call `i.next(k, v)` sets `k` and `v` to the key and value of the next pair and returns `TRUE`. If no entries remain, the call returns `FALSE` without setting `k` or `v`. It is a checked runtime error to call `next` or `seek` after `next` has returned `FALSE`. The client must ensure that while an iterator is in use, the parent table is not modified.

The call `i.seek(k)` skips past zero or more key-value pairs (either forward or backward) so that a subsequent call of `next` returns the first pair with key greater than or equal to `k` if `i` is in increasing order or with key less than or equal to `k` if `i` is in decreasing order.

The type `Default` is an implementation of `T` using randomized heap-ordered binary trees or "treaps" (see [1]). In this implementation, seeking forward (relative to the iterator's order) is more efficient than seeking backward. If a forward seek skips over `d` key-value pairs, the expected time for the seek is `O(log d)`. The time for a backward seek is `O(log(table.size()))`, no matter how far back it skips.

The call `dflt.init()` returns `dflt` after initializing it to an empty table.

The call `dflt.keyCompare(k1, k2)` returns `Key.Compare(k1, k2)`. The other methods call `keyCompare` whenever they need to consult `le(tbl)`. This means a subtype of `Default` can determine `le(tbl)` by overriding `keyCompare`, providing `keyCompare` implements a total order.

For efficiency, sorted tables and their iterators are not monitored, so a client accessing a table from multiple threads must ensure that if two operations are active concurrently, then neither of them has side effects on the same table or iterator. The `T.put`, `T.delete`, and `Default.init` methods are the only ones with side effects on the table. An iterator's `next` method has side-effects on the iterator.

The standard instances are named `SortedxyTbl`, for all combinations of `x` and `y` in the set `Atom, Int, Ref, Text`. The instances with `x = Atom` and `x = Ref` are useful only if you define a subtype overriding the `keyCompare` method.

3.7 Bundle

A `Bundle.T`, or bundle, is a collection of named byte string values, where the names and values are represented as `TEXTs`. The usefulness of bundles stems from the existence of a program called `m3bundle`. This program accepts an arbitrary set of files and produces the source code of a Modula-3 procedure that, when compiled and executed, returns a bundle containing the contents of the original files.

```
INTERFACE Bundle;

TYPE T <: REFANY;

PROCEDURE Get(b: T; nm: TEXT): TEXT;
```

If an element of `b` *has the name* `nm`*, return its value. Otherwise, return* `NIL`*.*

```
END Bundle.
```

To call `Bundle.Get`, you need a value of type `Bundle.T`. Given a collection of files, the program `m3bundle` generates the source code of an interface (`.i3` file) and a module (`.m3` file) implementing that interface. The interface contains a single procedure returning a bundle.

If you want to build a bundle with elements named `e1, ..., en` corresponding to values currently in files with pathnames `p1, ..., pN`, you invoke `m3bundle` as follows:

```
m3bundle -name Foo [-element e1 p1]...
```

`m3bundle` then produces an interface `Foo.i3` with this format:

```
INTERFACE Foo;
IMPORT Bundle;
PROCEDURE Get(): Bundle.T;
END Foo.
```

The call `Foo.Get()` returns a bundle `b` such that the call `Bundle.Get(b, nm)` returns the contents of file `pi` at the time `m3bundle` was invoked if `nm` equals one of the `ei` passed to `m3bundle`. Otherwise, `Bundle.Get(b, nm)` returns `NIL`.

For more information about `m3bundle`, consult its man page or other system-specific documentation.

4 Algorithms

4.1 ArraySort

```
GENERIC INTERFACE ArraySort(Elem);
```

Where `Elem.T` is a type that is not an open array type and `Elem` contains

```
PROCEDURE Compare(a, b: Elem.T): [-1 .. 1];
```

`Compare` must define a total order. Any parameter mode may be used.

```
PROCEDURE Sort(VAR a: ARRAY OF Elem.T; cmp := Elem.Compare);
```

Sort the elements of `a` using the order defined by `cmp`.

```
END ArraySort.
```

`Sort(a, cmp)` permutes the elements of `a` such that:

```
FIRST(a) <= i < j <= LAST(a)
```

implies

```
cmp(a[i], a[j]) <= 0.
```

The algorithm used is QuickSort:

- It is not stable.
- On average, it requires `O(N ln N)` comparison and assignment operations. In the worst case it may require `O(N*N)` operations.

For an expanded description of QuickSort, see [16].

The standard instances are named `IntArraySort` and `TextArraySort`. (There are no instances for `Atom.T` or `REFANY` since these types don't have a standard total order.)

4.2 Random

A `Random.T` (or just a generator) is a pseudo-random number generator.

```
INTERFACE Random;

TYPE
  T = OBJECT METHODS
    integer(min := FIRST(INTEGER);
      max := LAST(INTEGER)): INTEGER;
```

```
    real(min := 0.0e+0; max := 1.0e+0): REAL;
    longreal(min := 0.0d+0; max := 1.0d+0): LONGREAL;
    extended(min := 0.0x+0; max := 1.0x+0): EXTENDED;
    boolean(): BOOLEAN
  END;
  Default <: T OBJECT METHODS
    init(fixed := FALSE): Default
  END;
END Random.
```

Individual generators are unmonitored, and all the operations have side effects.

The methods provided by a generator `rand` are:

The call `rand.integer(a, b)` returns a uniformly distributed `INTEGER` in the closed interval `[a..b]`.

The call `rand.real(a, b)` returns a uniformly distributed `REAL` in the half-open interval `[a..b)`.

The call `longreal` and `extended` are like `real`, but return values of the specified types.

The call `rand.boolean()` returns a random `BOOLEAN` value.

It is a checked runtime error if `min > max` on any call.

`NEW(Default).init()` creates and initializes a generator (see below for implementation details). If `fixed` is `TRUE`, a predetermined sequence is used. If `fixed` is `FALSE`, `init` chooses a random seed in such a way that different sequences result even if `init` is called many times in close proximity.

Example. A good pseudo-random permutation of an array `a` can be generated as follows:

```
WITH rand = NEW(Random.Default).init() DO
  FOR i := FIRST(a) TO LAST(a) - 1 DO
    WITH j = rand.integer(i, LAST(a)) DO
      Exchange a[i] and a[j]
    END
  END
END
```

SRC Modula-3 implementation details. The object returned by a call of `New(Default).init` uses an additive generator based on Knuth's Algorithm 3.2.2A (see [9]).

4.3 Fingerprint

A `Fingerprint.T` is a 64-bit checksum. This interface provides procedures that can be used to fingerprint text strings or more general data structures, such as graphs.

The interface is based on the original idea of M. O. Rabin [14], as refined by Andrei Broder [3].

```
INTERFACE Fingerprint;

TYPE T = RECORD
    byte: ARRAY [0..7] OF BITS 8 FOR [0..255]
  END;

PROCEDURE FromText(txt: TEXT): T;
```

Return the fingerprint of `txt`.

```
PROCEDURE Combine(READONLY fp1, fp2: T): T;
```

Return the fingerprint of the ordered pair `(fp1, fp2)`.

```
CONST Zero = T{ARRAY [0..7] OF BITS 8 FOR [0..255] {0, ..}};

VAR (*CONST*) OfEmpty: T;
```

The fingerprint of the empty text.

The following procedure, `FromChars`, provides two additional features. First, it takes an array of characters instead of a `TEXT`, which can save on allocations. Second, it can be used to compute the fingerprint of a sequence incrementally, a buffer at a time, since it accepts the checksum of the previous text together with a new buffer full of text and computes the checksum of the whole text.

```
PROCEDURE FromChars
  (READONLY buff: ARRAY OF CHAR; READONLY fp: T): T;
```

Return the fingerprint of `t & Text.FromChars(buff)`, *where* `t` *is the text whose fingerprint is* `fp`.

The last two procedures in the interface allow you to use fingerprints as the key type in a generic table.

```
PROCEDURE Equal(READONLY fp1, fp2: T): BOOLEAN;
```

Return `fp1 = fp2`.

```
PROCEDURE Hash(READONLY fp: T): INTEGER;
```

Return a hash code for `fp`.

```
END Fingerprint.
```

The probabilistic guarantee. The fingerprint module produces a provably secure checksum. To explain exactly what this means requires a few definitions.

Define a *nest* to be a text string or an ordered pair of two nests. The fingerprint `FP(x)` of a nest `x` is defined as follows:

> `FP(x) = FromText(x)` if `x` is a text
> `FP(x) = Combine(FP(y), FP(z))` if `x` is a pair `(y, z)`.

Two nests `x` and `y` *collide* if `x # y` but `FP(x) = FP(y)`. (Two texts are equal if they are `Text.Equal`, and two pairs are equal if their corresponding components are equal. We assume that nests are finite and non-circular.)

A nest `x` is a *subnest* of `y` if it occurs anywhere in `y`; that is, if it equals `y` or if `y` is an ordered pair and `x` is a subnest of one of `y`'s components.

Define the *length* of a nest to be the sum of the lengths of all the distinct texts that occur anywhere inside it, and the *size* of a nest to be the number of distinct subnests that it has. For example, the length of the nest

```
(("a", "b"), ("a", "b"))
```

is two, since the only texts that occur inside it are `a` and `b`, whose lengths sum to two. The size of the nest is four, since its distinct subnests are itself, the pair (`a`, `b`), and the texts `a` and `b`.

The fingerprint module contains a magic number that was chosen on 12 December 1986 by flipping a quarter 128 times in Andrei Broder's office at SRC. The checksum produced by the package is a function of this magic number.

The probabilistic guarantee for the fingerprint algorithm is that for any nest `S`, even one produced by an adversary who knows everything about the algorithm except the magic number, the probability that the 1986 coin-flipping produced a magic number such that some pair of subnests of `S` collide is at most

```
(length(S) * size(S)) / 2^62.
```

From this basic guarantee you can compute an upper bound on the probability of a collision in your application. For example, if two texts `t1` and `t2` collide, then the nest `(t1, t2)` contains two colliding subnests. The odds against this are at least `2^62` to `N * 3`, where `N` is the total length of the two texts. For example, if the total length is a million characters, the collision probability is at most

```
(10^6 * 3) / 2^62
```

This is less than one in a trillion.

Similarly, given a thousand texts each of length a thousand, considering the linear list of all of them as a nest and applying the guarantee, we conclude that the probability that some pair collide is at most

```
(10^6 * 2 * 10^3) / 2^62
```

which is less than one in `2^31`, or less than one in `10^9`.

Of course these are probabilities with respect to a random coin-flipping that has already happened and is therefore not random anymore. If you were present in Andrei's office, or if you look at the magic number in the implementation, you can easily construct a small nest that contains a collision. The probabilistic guarantee is valid only if the structure you are fingerprinting is independent of the coin-flipping event. For example, it would not really be a good idea to fingerprint the text of the module `Fingerprint.m3`, since that text contains the magic number as a constant, and therefore the probabilistic guarantee says nothing about the quality of its fingerprint.

Example applications. Fingerprints are useful in many aspects of computer systems. For example, to determine if two long files stored on different computer systems are identical, it is not necessary to transfer the entire file from one system to another: it suffices to fingerprint the files and transfer and compare the fingerprints. (Assuming that the probabilistic guarantee is good enough for your application.)

Fingerprints are also a key technology for achieving type safety in distributed programming. Within a single address space, the compiler and linker can ensure that the value of every variable is consistent with its type. In a distributed computation, where values in one program are reduced to bit sequences and sent over the network to become values of variables in another program, the compiler cannot perform this check: whatever the compiler does, a programmer could erroneously change the type in one of the programs and recompile and execute it. Some kind of runtime check is required when the value is transferred. The simplest check is to send the type of the value along with the value itself, and then to check the type when the value is received. But types can be quite complicated in modern programming languages, and it would be inefficient to communicate types by sending a full description of their structure over the wire. Fingerprints provide the answer: the sending program computes a fingerprint of the type, and the receiving program compares the fingerprint with the fingerprint of the receiving variable. Fingerprints play essentially the same role in making persistent storage typesafe. The SRC Modula-3 runtime provides an interface for converting between typecodes and type fingerprints, for exactly this purpose.

Fingerprinting general data structures. The `Combine` function makes it convenient to fingerprint many data structures. For example, consider a directed acyclic graph (DAG) in which each node `nd` has a text label `lbl(nd)` and `deg(nd)` neighbor nodes `nd[1]`, ..., `nd[deg(nd)]`. Such a graph represents an expression in which a node `nd` of degree zero represents a constant value named by `lbl(nd)`, and a node `nd` of degree greater than zero represents an expression with root operator `lbl(nd)` and arguments `nd[1]`, ..., `nd[deg(nd)]`.

One way to find common subexpressions is to compute a fingerprint `F(nd)` for every node `nd` by the following rule:

```
PROCEDURE F(nd): T =
  VAR res := FromText(lbl(nd)); BEGIN
    FOR i := 1 TO deg(nd) DO
      res := Combine(res, F(nd[i]))
    END;
    RETURN res
  END F;
```

(If the DAG is not a tree, the program as written will recompute the fingerprint of nodes with multiple parents, possibly many times. To avoid this, you can easily modify the program to record the fingerprint in the node, so that the total computation time is proportional to the size of the graph.)

The procedure `F` has the property that with high probability, two nodes have the same fingerprint if and only if they represent common subexpressions. This is a consequence of the probabilistic guarantee together with the observation that `f(a1, ..., an)` and `g(b1, ..., bm)` are common subexpressions if and only if the nests

```
( ... ((f, a1),  a2), ... an)
( ... ((g, b1),  b2), ... bm)
```

are equal.

Other data structures, such as cyclic graphs, can be fingerprinted with more elaborate strategies based on the same idea. When designing fingerprinting algorithms for other data structures, it is important to remember that `Combine` is neither commutative nor associative.

Pitfalls. The original fingerprint interface offered at SRC did not include the procedure `Combine`. The Vesta configuration management project built a system that cached intermediate results for large software builds. Abstractly, this is a special case of the common subexpression problem mentioned previously, and the project used fingerprints as keys in the cache. It is instructive to learn what happened.

You might think that a simple way to solve the common subexpression problem without `Combine` would be to fingerprint the texts that result from printing the expressions represented by the nodes of the DAG. But if the DAG is not a tree, this is a serious error, since the length of the strings produced by printing a DAG can grow geometrically with its size, and therefore the probabilistic guarantee becomes useless even for quite small DAGs.

Avoiding this error, the Vesta group computed the fingerprint of a node by concatenating the node's label with the *fingerprints* of its children—treating these fingerprints as 8-byte texts— and fingerprinted the resulting text. With this strategy, the number of texts fingerprinted is proportional to the number of nodes of the DAG, and the total length of these texts is proportional to the number of edges of the DAG. Thus the method appears efficient and sound.

Alas, the method is not sound. Recall that the probabilistic guarantee is valid only if the strings being fingerprinted are independent of the magic number. But fingerprints themselves are dependent on the magic number, so the probabalistic guarantee is invalid whenever fingerprints are fingerprinted. The Vesta group was soon debugging an unexpected collision.

The moral is simple: the procedure `Combine` is a convenience, but it is also much more than a convenience. It should be the only way that you ever generate a fingerprint from another fingerprint. In particular, never treat a fingerprint as text to be passed to `FromText`.

5 I/O Streams

The interfaces `Wr` and `Rd` provide object-oriented output and input streams, called *writers* and *readers*. These were invented by Stoy and Strachey in 1972 [18]. The versions presented here are slight modifications of the versions in the first Modula-3 book [13]. We also present related interfaces for obtaining writers and readers connected to texts or files (`TextWr`, `TextRd`, `FileWr`, and `FileRd`), for accessing standard streams (`Stdio`), and for performing I/O in simple programs (`IO`). The interfaces `FileWr` and `FileRd` supersede the interface `FileStream` in [13].

5.1 IO

The `IO` interface provides textual input and output for simple programs. For more detailed control, use the interfaces `Rd`, `Wr`, `Stdio`, `FileRd`, `FileWr`, `Fmt`, and `Lex`.

The input procedures take arguments of type `Rd.T` that specify which input stream to use. If this argument is defaulted, standard input (`Stdio.stdin`) is used. Similarly, if an argument of type `Wr.T` to an output procedure is defaulted, `Stdio.stdout` is used.

```
INTERFACE IO;

IMPORT Rd, Wr;

PROCEDURE Put(txt: TEXT; wr: Wr.T := NIL);
```

Output `txt` *to* `wr` *and flush* `wr`*.*

```
PROCEDURE PutInt(n: INTEGER; wr: Wr.T := NIL);
```

Output `Fmt.Int(n)` *to* `wr` *and flush* `wr`*.*

```
PROCEDURE PutReal(r: REAL; wr: Wr.T := NIL);
```

Output `Fmt.Real(r)` *to* `wr` *and flush* `wr`*.*

```
PROCEDURE EOF(rd: Rd.T := NIL): BOOLEAN;
```

Return `TRUE` *iff* `rd` *is at end-of-file.*

```
EXCEPTION Error;
```

The exception `Error` is raised whenever a `Get` procedure encounters syntactically invalid input, including unexpected end-of-file.

```
PROCEDURE GetLine(rd: Rd.T := NIL): TEXT RAISES {Error};
```

Read a line of text from `rd` *and return it.*

A line of text is either zero or more characters terminated by a line break, or one or more characters terminated by an end-of-file. In the former case, `GetLine` consumes the line break but does not include it in the returned value. A line break is either `"\n"` or `"\r\n"`.

```
PROCEDURE GetChar(rd: Rd.T := NIL): CHAR RAISES {Error};
```

Read the next character from `rd` and return it.

```
PROCEDURE GetInt(rd: Rd.T := NIL): INTEGER RAISES {Error};
```

Read a decimal numeral from `rd` using `Lex.Int` and return its value.

```
PROCEDURE GetReal(rd: Rd.T := NIL): REAL RAISES {Error};
```

Read a real number from `rd` using `Lex.Real` and return its value.

```
PROCEDURE OpenRead(f: TEXT): Rd.T;
```

Open the file name `f` for reading and return a reader on its contents. If the file doesn't exist or is not readable, return `NIL`.

```
PROCEDURE OpenWrite(f: TEXT): Wr.T;
```

Open the file named `f` for writing and return a writer on its contents. If the file does not exist it will be created. If the process does not have the authority to modify or create the file, return `NIL`.

```
END IO.
```

5.2 Wr

A `Wr.T` (or "writer") is a character output stream. The basic operation on a writer is `PutChar`, which extends a writer's character sequence by one character. Some writers (called "seekable writers") also allow overwriting in the middle of the sequence. For example, writers to random access files are seekable, but writers to terminals and sequential files are not.

Writers can be (and usually are) buffered. This means that operations on the writer don't immediately affect the underlying target of the writer, but are saved up and performed later. For example, a writer to a disk file is not likely to update the disk after each character.

Abstractly, a writer `wr` consists of:

`len(wr)`	a non-negative integer
`c(wr)`	a character sequence of length `len(wr)`
`cur(wr)`	an integer in the range `[0..len(wr)]`
`target(wr)`	a character sequence

`closed(wr)`	a boolean
`seekable(wr)`	a boolean
`buffered(wr)`	a boolean

These values are generally not directly represented in the data fields of a writer object, but in principle they determine the state of the writer.

The sequence `c(wr)` is zero-based: `c(wr)[i]` is valid for `i` from 0 through `len(wr)-1`. The value of `cur(wr)` is the index of the character in `c(wr)` that will be replaced or appended by the next call to `PutChar`. If `wr` is not seekable, then `cur(wr)` is always equal to `len(wr)`, since in this case all writing happens at the end.

The difference between `c(wr)` and `target(wr)` reflects the buffering: if `wr` is not buffered, then `target(wr)` is updated to equal `c(wr)` after every operation; if `wr` is buffered, then updates to `target(wr)` can be delayed. For example, in a writer to a file, `target(wr)` is the actual sequence of characters on the disk; in a writer to a terminal, `target(wr)` is the sequence of characters that have actually been transmitted. (This sequence may not exist in any data structure, but it still exists abstractly.)

If `wr` is buffered, then the assignment `target(wr) := c(wr)` can happen asynchronously at any time, although the procedures in this interface are atomic with respect to such assignments.

Every writer is a monitor; that is, it contains an internal lock that is acquired and held for each operation in this interface, so that concurrent operations will appear atomic. For faster, unmonitored access, see the `UnsafeWr` interface.

If you are implementing a long-lived writer class, such as a pipe or TCP stream, the index of the writer may eventually overflow, causing the program to crash with a bounds fault. We recommend that you provide an operation to reset the writer index, which the client can call periodically.

It is useful to specify the effect of several of the procedures in this interface in terms of the action `PutC(wr, ch)`, which outputs the character `ch` to the writer `wr`:

```
PutC(wr, ch) =
  IF closed(wr) THEN Cause checked runtime error END;
  IF cur(wr) = len(wr) THEN
    Extend c(wr) by one character, incrementing len(wr)
  END;
  c(wr)[cur(wr)] := ch;
  INC(cur(wr));
```

`PutC` is used only in specifying the interface; it is not a real procedure.

```
INTERFACE Wr;

IMPORT AtomList;
FROM Thread IMPORT Alerted;
```

```
TYPE T <: ROOT;

EXCEPTION Failure(AtomList.T);
```

Since there are many classes of writers, there are many ways that a writer can break—for example, the network can go down, the disk can fill up, etc. All problems of this sort are reported by raising the exception `Failure`. The documentation of each writer class should specify what failures the class can raise and how they are encoded in the argument to `Failure`.

Illegal operations (for example, writing to a closed writer) cause checked runtime errors.

```
VAR (*CONST*) EOL: TEXT;
```

End of line.

On POSIX, `EOL` is `"\n"`; on Win32, `EOL` is `"\r\n"`.

```
PROCEDURE PutChar(wr: T; ch: CHAR) RAISES {Failure, Alerted};
```

Output `ch` *to* `wr`. *More precisely, this is equivalent to:*

```
PutC(wr, ch); IF NOT buffered(wr) THEN Flush(wr) END
```

Many operations on a writer can wait indefinitely. For example, `PutChar` can wait if the user has suspended output to his terminal. These waits can be alertable, so each procedure that might wait includes `Thread.Alerted` in its raises clause.

```
PROCEDURE PutText(wr: T; t: TEXT) RAISES {Failure, Alerted};
```

Output `t` *to* `wr`. *More precisely, this is equivalent to:*

```
FOR i := 0 TO Text.Length(t) - 1 DO
  PutC(wr, Text.GetChar(t, i))
END;
IF NOT buffered(wr) THEN Flush(wr) END
```

except that, like all operations in this interface, it is atomic with respect to other operations in the interface. (It would be wrong to write `PutChar` instead of `PutC`, since `PutChar` always flushes if the writer is unbuffered.)

```
PROCEDURE PutString(wr: T; READONLY a: ARRAY OF CHAR)
  RAISES {Failure, Alerted};
```

Output `a` *to* `wr`. *More precisely, other than the fact that this is atomic, it is equivalent to:*

```
FOR i := FIRST(a) TO LAST(a) DO PutC(wr, a[i]) END;
IF NOT buffered(wr) THEN Flush(wr) END
```

```
PROCEDURE Seek(wr: T; n: CARDINAL) RAISES {Failure, Alerted};
```

Set the current position of `wr` *to* `n`*. This is an error if* `wr` *is closed. More precisely, this is equivalent to:*

```
IF wr.closed OR NOT seekable(wr) THEN
  Cause checked runtime error
END;
cur(wr) := MIN(n, len(wr))
```

```
PROCEDURE Flush(wr: T) RAISES {Failure, Alerted};
```

Perform all buffered operations. That is, set `target(wr) := c(wr)`*. It is a checked runtime error if* `wr` *is closed.*

```
PROCEDURE Close(wr: T) RAISES {Failure, Alerted};
```

Flush `wr`, release any resources associated with `wr`, and set `closed(wr) := TRUE`. The documentation for a procedure that creates a writer should specify what resources are released when the writer is closed. This leaves `closed(wr)` equal to `TRUE` even if it raises an exception, and is a no-op if `wr` is closed.

```
PROCEDURE Length(wr: T): CARDINAL RAISES {Failure, Alerted};
PROCEDURE Index(wr: T): CARDINAL RAISES {};
PROCEDURE Seekable(wr: T): BOOLEAN RAISES {};
PROCEDURE Closed(wr: T): BOOLEAN RAISES {};
PROCEDURE Buffered(wr: T): BOOLEAN RAISES {};
```

These procedures return `len(wr)`, `cur(wr)`, `seekable(wr)`, `closed(wr)`, *and* `buffered(wr)`*, respectively.* `Length` *and* `Index` *cause a checked runtime error if* `wr` *is closed; the other three procedures do not.*

```
END Wr.
```

5.3 Rd

An `Rd.T` (or "reader") is a character input stream. The basic operation on a reader is `GetChar`, which returns the source character at the "current position" and advances the current position by one. Some readers are "seekable", which means that they also allow setting the current position anywhere in the source. For example, readers from random access files are seekable; readers from terminals and sequential files are not.

Some readers are "intermittent", which means that the source of the reader trickles in rather than being available to the implementation all at once. For example, the input stream from an interactive terminal is intermittent. An intermittent reader is never seekable.

Abstractly, a reader `rd` consists of

`len(rd)`	the number of source characters
`src(rd)`	a sequence of length `len(rd)+1`
`cur(rd)`	an integer in the range `[0..len(rd)]`
`avail(rd)`	an integer in the range `[cur(rd)..len(rd)+1]`
`closed(rd)`	a boolean
`seekable(rd)`	a boolean
`intermittent(rd)`	a boolean

These values are not necessarily directly represented in the data fields of a reader object. In particular, for an intermittent reader, `len(rd)` may be unknown to the implementation. But in principle the values determine the state of the reader.

The sequence `src(rd)` is zero-based: `src(rd)[i]` is valid for `i` from 0 to `len(rd)`. The first `len(rd)` elements of `src` are the characters that are the source of the reader. The final element is a special value `eof` used to represent end-of-file. The value `eof` is not a character.

The value of `cur(rd)` is the index in `src(rd)` of the next character to be returned by `GetChar`, unless `cur(rd) = len(rd)`, in which case a call to `GetChar` will raise the exception `EndOfFile`.

The value of `avail(rd)` is important for intermittent readers: the elements whose indexes in `src(rd)` are in the range `[cur(rd)..avail(rd)-1]` are available to the implementation and can be read by clients without blocking. If the client tries to read further, the implementation will block waiting for the other characters. If `rd` is not intermittent, then `avail(rd)` is equal to `len(rd)+1`. If `rd` is intermittent, then `avail(rd)` can increase asynchronously, although the procedures in this interface are atomic with respect to such increases.

The definitions above encompass readers with infinite sources. If `rd` is such a reader, then `len(rd)` and `len(rd)+1` are both infinity, and there is no final `eof` value.

Every reader is a monitor; that is, it contains an internal lock that is acquired and held for each operation in this interface, so that concurrent operations will appear atomic. For faster, unmonitored access, see the `UnsafeRd` interface.

If you are implementing a long-lived reader class, such as a pipe or TCP stream, the index of the reader may eventually overflow, causing the program to crash with a bounds fault. We recommend that you provide an operation to reset the reader index, which the client can call periodically.

```
INTERFACE Rd;

IMPORT AtomList;
FROM Thread IMPORT Alerted;

TYPE T <: ROOT;
```

```
EXCEPTION EndOfFile; Failure(AtomList.T);
```

Since there are many classes of readers, there are many ways that a reader can break—for example, the connection to a terminal can be broken, the disk can signal a read error, etc. All problems of this sort are reported by raising the exception `Failure`. The documentation of a reader class should specify what failures the class can raise and how they are encoded in the argument to `Failure`.

Illegal operations cause a checked runtime error.

```
PROCEDURE GetChar(rd: T): CHAR
  RAISES {EndOfFile, Failure, Alerted};
```

Return the next character from `rd`*. More precisely, this is equivalent to the following, in which* `res` *is a local variable of type* `CHAR`*:*

```
IF closed(rd) THEN Cause checked runtime error END;
Block until avail(rd) > cur(rd);
IF cur(rd) = len(rd) THEN
  RAISE EndOfFile
ELSE
  res := src(rd)[cur(rd)]; INC(cur(rd)); RETURN res
END
```

Many operations on a reader can wait indefinitely. For example, `GetChar` can wait if the user is not typing. In general these waits are alertable, so each procedure that might wait includes `Thread.Alerted` in its `RAISES` clause.

```
PROCEDURE EOF(rd: T): BOOLEAN RAISES {Failure, Alerted};
```

Return `TRUE` *iff* `rd` *is at end-of-file. More precisely, this is equivalent to:*

```
IF closed(rd) THEN Cause checked runtime error END;
Block until avail(rd) > cur(rd);
RETURN cur(rd) = len(rd)
```

Notice that on an intermittent reader, `EOF` can block. For example, if there are no characters buffered in a terminal reader, `EOF` must wait until the user types one before it can determine whether he typed the special key signalling end-of-file. If you are using `EOF` in an interactive input loop, the right sequence of operations is:

1. prompt the user;
2. call `EOF`, which probably waits on user input;
3. presuming that `EOF` returned `FALSE`, read the user's input.

```
PROCEDURE UnGetChar(rd: T) RAISES {};
```

"Push back" the last character read from `rd`*, so that the next call to* `GetChar` *will read it again. More precisely, this is equivalent to the following:*

```
IF closed(rd) THEN Cause checked runtime error END;
IF cur(rd) > 0 THEN DEC(cur(rd)) END
```

except there is a special rule: `UnGetChar(rd)` is guaranteed to work only if `GetChar(rd)` was the last operation on `rd`. Thus `UnGetChar` cannot be called twice in a row, or after `Seek` or `EOF`. If this rule is violated, the implementation is allowed (but not required) to cause a checked runtime error.

```
PROCEDURE CharsReady(rd: T): CARDINAL RAISES {Failure};
```

Return some number of characters that can be read without indefinite waiting. The "end of file marker" counts as one character for this purpose, so `CharsReady` *will return 1, not 0, if* `EOF(rd)` *is true. More precisely, this is equivalent to the following:*

```
IF closed(rd) THEN Cause checked runtime error END;
IF avail(rd) = cur(rd) THEN
  RETURN 0
ELSE
  RETURN some number in the range [1 .. avail(rd) - cur(rd)]
END;
```

Warning: `CharsReady` can return a result less than `avail(rd) - cur(rd)`; also, more characters might trickle in just as `CharsReady` returns. So the code to flush buffered input without blocking requires a loop:

```
LOOP
  n := Rd.CharsReady(rd);
  IF n = 0 THEN EXIT END;
  FOR i := 1 TO n DO EVAL Rd.GetChar(rd) END
END;
```

```
PROCEDURE GetSub(rd: T; VAR (*OUT*) str: ARRAY OF CHAR)
  : CARDINAL RAISES {Failure, Alerted};
```

Read from `rd` *into* `str` *until* `rd` *is exhausted or* `str` *is filled. More precisely, this is equivalent to the following, in which* `i` *is a local variable:*

```
i := 0;
WHILE i # NUMBER(str) AND NOT EOF(rd) DO
  str[i] := GetChar(rd); INC(i)
END;
```

```
  RETURN i
```

```
PROCEDURE GetSubLine(rd: T; VAR (*OUT*) str: ARRAY OF CHAR)
  : CARDINAL RAISES {Failure, Alerted};
```

Read from `rd` into `str` until a newline is read, `rd` is exhausted, or `str` is filled. More precisely, this is equivalent to the following, in which `i` is a local variable:

```
  i := 0;
  WHILE
    i # NUMBER(str) AND
    (i = 0 OR str[i-1] # '\n') AND
    NOT EOF(rd)
  DO
    str[i] := GetChar(rd); INC(i)
  END;
  RETURN i
```

Note that `GetLine` strips the terminating line break, while `GetSubLine` does not.

```
PROCEDURE GetText(rd: T; len: CARDINAL): TEXT
  RAISES {Failure, Alerted};
```

Read from `rd` until it is exhausted or `len` characters have been read, and return the result as a `TEXT`. More precisely, this is equivalent to the following, in which `i` and `res` are local variables:

```
  res := ""; i := 0;
  WHILE i # len AND NOT EOF(rd) DO
    res := res & Text.FromChar(GetChar(rd));
    INC(i)
  END;
  RETURN res
```

```
PROCEDURE GetLine(rd: T): TEXT
  RAISES {EndOfFile, Failure, Alerted};
```

If `EOF(rd)` then raise `EndOfFile`. Otherwise, read characters until a line break is read or `rd` is exhausted, and return the result as a `TEXT`—but discard the line break if it is present. A line break is either `"\n"` or `"\r\n"` More precisely, this is equivalent to the following, in which `ch` and `res` are local variables:

```
  IF EOF(rd) THEN RAISE EndOfFile END;
  res := ""; ch := '\000'; (* any char but newline *)
```

```
    WHILE ch # '\n' AND NOT EOF(rd) DO
      ch := GetChar(rd);
      IF ch = '\n' THEN
        IF NOT Text.Empty(res) AND
            Text.GetChar(res, Text.Length(res)-1) = '\r' THEN
          res := Text.Sub(res, 0, Text.Length(res)-1)
        END
      ELSE
        res := res & Text.FromChar(ch)
      END
    RETURN res
```

```
PROCEDURE Seek(rd: T; n: CARDINAL) RAISES {Failure, Alerted};
```

This is equivalent to:

```
    IF closed(rd) OR NOT seekable(rd) THEN
      Cause checked runtime error
    END;
    cur(rd) := MIN(n, len(rd))
```

```
PROCEDURE Close(rd: T) RAISES {Failure, Alerted};
```

Release any resources associated with `rd` *and set* `closed(rd) := TRUE`*. The documentation of a procedure that creates a reader should specify what resources are released when the reader is closed. This leaves* `rd` *closed even if it raises an exception, and is a no-op if* `rd` *is closed.*

```
PROCEDURE Index(rd: T): CARDINAL RAISES {};
```

This is equivalent to:

```
    IF closed(rd) THEN Cause checked runtime error END;
    RETURN cur(rd)
```

```
PROCEDURE Length(rd: T): INTEGER RAISES {Failure, Alerted};
```

This is equivalent to:

```
    IF closed(rd) THEN
      Cause checked runtime error
    END;
    RETURN len(rd)
```

If `len(rd)` is unknown to the implementation of an intermittent reader, `Length(rd)` returns -1.

```
PROCEDURE Intermittent(rd: T): BOOLEAN RAISES {};
```

```
PROCEDURE Seekable(rd: T): BOOLEAN RAISES {};
PROCEDURE Closed(rd: T): BOOLEAN RAISES {};
```

Return `intermittent(rd)`, `seekable(rd)`, and `closed(rd)`, respectively. These can be applied to closed readers.

```
END Rd.
```

5.4 TextWr and TextRd

A `TextWr.T`, or text writer, is a writer the contents of whose internal buffer can be retrieved as a `TEXT`. Retrieving the buffer resets the target to be empty. Text writers are buffered, seekable, and never raise `Failure` or `Alerted`. The fact that they are buffered is essentially unobservable, since there is no way for the client to access the target except through the text writer.

```
INTERFACE TextWr;

IMPORT Wr;

TYPE
  T <: Public;
  Public = Wr.T OBJECT METHODS init(): T END;
```

The call `wr.init()` initializes `wr` to be a seekable writer with `c(wr)` set to the empty sequence and `cur(wr)` to 0. The writer has no upper bound on its length.

```
PROCEDURE New(): T;
```

Equivalent to `NEW(T).init()`.

```
PROCEDURE ToText(wr: T): TEXT;
```

Return `c(wr)`, resetting `c(wr)` to the empty sequence and `cur(wr)` to 0.

```
END TextWr.
```

A `TextRd.T`, or text reader, is a reader that delivers the characters of a `TEXT` supplied when the reader was created. Text readers are seekable, non-intermittent, and never raise `Failure` or `Alerted`.

```
INTERFACE TextRd;

IMPORT Rd;

TYPE
  T <: Public;
  Public = Rd.T OBJECT METHODS init(t: TEXT): T END;
```

The call `rd.init(t)` initializes `rd` as a seekable, non-intermittent reader with:

```
len(rd) = Text.Length(t)
src(rd) = characters of t
cur(rd) = 0
```

It is a checked runtime error if `t = NIL`.

```
PROCEDURE New(t: TEXT): T;
```

Equivalent to `NEW(T).init(t)`.

```
END TextRd.
```

5.5 Stdio, FileWr, and FileRd

`Stdio` provides streams for standard input, standard output, and standard error. These streams correspond to file handles returned by the `GetStandardHandles` procedure in the `Process` interface.

```
INTERFACE Stdio;

IMPORT Rd, Wr;

VAR
  stdin: Rd.T;
  stdout: Wr.T;
  stderr: Wr.T;
  bufferedStderr: Wr.T;

END Stdio.
```

The initialization of these streams depends on the underlying operating system.

If the standard error stream is directed to a terminal, it will be unbuffered, so that explicit `Wr.Flush` calls are unnecessary for interactive programs. A buffered version of the standard error stream is also provided, but programs should not use both `stderr` and `bufferedStderr`.

If the streams are directed to or from random-access files, they will be seekable.

It is possible that `stderr` is equal to `stdout`. Therefore, programs that perform seek operations on `stdout` should take care not to destroy output data when writing error messages.

A `FileWr.T`, or file writer, is a writer on a `File.T`.

```
INTERFACE FileWr;

IMPORT Wr, File, OSError, Pathname;
```

```
TYPE
  T <: Public;
  Public = Wr.T OBJECT METHODS
    init(h: File.T; buffered: BOOLEAN := TRUE): T
      RAISES {OSError.E}
  END;
```

If `w` *is a file writer and* `h` *is a file handle, the call* `w.init(h)` *initializes* `w` *so that characters output to* `w` *are written to* `h` *and so that closing* `w` *closes* `h`*.*

If `h` is a regular file handle and `b` is a Boolean, `w.init(h, b)` causes `w` to be a buffered seekable writer and initializes `cur(w)` to `cur(h)`.

For any other file handle `h`, `w.init(h, b)` causes `w` to be a nonseekable writer, buffered if and only if `b` is `TRUE`, and initializes `cur(w)` to zero.

If a subsequent writer operation on `w` raises `Wr.Failure`, the associated exception argument is the `AtomList.T` argument accompanying an `OSError.E` exception from a file operation on `h`.

```
PROCEDURE Open(p: Pathname.T): T RAISES {OSError.E};
```

Return a file writer whose target is the file named `p`*. If the file does not exist, it is created. If the file exists, it is truncated to a size of zero.*

The call `Open(p)` is equivalent to the following:

```
RETURN NEW(T).init(FS.OpenFile(p))
```

```
PROCEDURE OpenAppend(p: Pathname.T): T RAISES {OSError.E};
```

Return a file writer whose target is the file named `p`*. If the file does not exist, it is created. If the file exists, the writer is positioned to append to the existing contents of the file.*

The call `OpenAppend(p)` is equivalent to the following:

```
WITH h = FS.OpenFile(p, truncate := FALSE) DO
  EVAL h.seek(RegularFile.Origin.End, 0);
  RETURN NEW(T).init(h)
END
```

```
END FileWr.
```

A `FileRd.T`, or file reader, is a reader on a `File.T`.

```
INTERFACE FileRd;

IMPORT Rd, File, OSError, Pathname;
```

```
TYPE
  T <: Public;
  Public = Rd.T OBJECT METHODS
    init(h: File.T): T RAISES {OSError.E}
  END;
```

If `r` is a file reader and `h` is a file handle, the call `r.init(h)` initializes `r` so that reading `r` reads characters from `h`, and so that closing `r` closes `h`.

If `h` is a regular file handle, `r.init(h)` causes `r` to be a nonintermittent, seekable reader and initializes `cur(r)` to `cur(h)`.

For any other file handle `h`, `r.init(h)` causes `r` to be an intermittent, nonseekable reader and initializes `cur(r)` to zero.

If a subsequent reader operation on `r` raises `Rd.Failure`, the associated exception argument is the `AtomList.T` argument accompanying an `OSError.E` exception from a file operation on `h`.

```
PROCEDURE Open(p: Pathname.T): T RAISES {OSError.E};
```

Return a file reader whose source is the file named `p`. If the file does not exist, `OSError.E` is raised with an implementation-defined code.

The call `Open(p)` is equivalent to

```
RETURN NEW(T).init(FS.OpenFileReadonly(p))
```

```
END FileRd.
```

6 Operating System

The interfaces in this section provide access to operating system facilities: timekeeping, files, pathnames, directories, and processes. The interfaces are intended to be implementable at least on POSIX [8] and Win32 [12] systems.

6.1 Time

A `Time.T` represents a moment in time, reckoned as a number of seconds since some epoch or starting point.

```
INTERFACE Time;

TYPE T = LONGREAL;

PROCEDURE Now(): T;
```

Return the current moment in time.

```
VAR (*CONST*) Grain: LONGREAL;
```

If a thread performs `t0 := Time.Now(); t1 := Time.Now()`, then either `t1 = t0` or `t1 >= t0 + Time.Grain`. `Grain` always lies in the half-open interval `(0..1]` and is usually no larger than one sixtieth of a second.

```
END Time.
```

There are a variety of timekeeping needs, and `Time.Now` may not satisfy all of them. It is intended to be useful for recording times and measuring intervals arising during the execution of computer programs with a resolution comparable to human reaction times.

The epoch for a `Time.T` varies from one operating system to another. To determine the epoch, call `Date.FromTime(0.0D0, Date.UTC)`. Note that communicating a `Time.T` between systems, say via remote procedure call or pickles, is likely to be a bad idea.

In many computers, `Time.Now` is implemented with the technology of an inexpensive wristwatch, and is therefore likely to suffer from similar errors: the rate may vary, and the value may be changed by a human operator.

The `Thread` interface contains procedures that delay the execution of the calling thread for a specified duration. The `Tick` interface provides access to a clock with subsecond resolution.

6.2 Date

A `Date.T` is a moment in time, expressed according to the standard (Gregorian) calendar, as observed in some time zone. A `Date.TimeZone` (or just a time zone) is an object that encapsulates the rules for converting from UTC (universal coordinated time, sometimes known as Greenwich mean time) to local time within a particular jurisdiction, taking into account daylight time when appropriate.

```
INTERFACE Date;

IMPORT Time;

TYPE
  T = RECORD
    year: CARDINAL; (* e.g., 1992 *)
    month: Month;
    day: [1 .. 31];
    hour: [0 .. 23];
    minute: [0 .. 59];
    second: [0 .. 59];
    offset: INTEGER;
    zone: TEXT;
    weekDay: WeekDay
  END;
  Month = {Jan, Feb, Mar, Apr, May, Jun, Jul,
           Aug, Sep, Oct, Nov, Dec};
  WeekDay = {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

A date's `offset` field specifies the difference in the readings of two clocks, one set to UTC and one set to local time, at the moment the date occurred, and thus reflects daylight time when appropriate. This difference is specified in seconds, with positive values corresponding to local zones behind (west of) UTC. A date's `zone` field specifies a name (often a three-letter abbreviation) for the time zone in which the date is observed, for example, "PDT" for Pacific Daylight Time.

```
TYPE TimeZone <: REFANY;

VAR Local, UTC: TimeZone;
```

`Local` is initialized to the time zone in which the computer running this program is located. `UTC` is initialized to the time zone for universal coordinated time.

```
PROCEDURE FromTime(t: Time.T; z: TimeZone := NIL): T;
```

Return the date corresponding to `t`, as observed in the time zone `z`. If `z` is `NIL`, `Local` is used.

```
EXCEPTION Error;

PROCEDURE ToTime(READONLY d: T): Time.T RAISES {Error};
```

Return the time corresponding to the date `d`*, using the field* `offset` *rather than* `zone` *and ignoring the field* `weekDay`*. Raise* `Error` *if* `d` *cannot be represented as a* `Time.T`*.*

```
END Date.
```

On POSIX systems, `FromTime(t, Local)` calls `localtime(3)`. On Win32 systems, it calls `GetTimeZoneInformation`. Some systems keep local time instead of UTC, and typically don't record the identity of the local time zone. On such a system, `FromTime(t, Local)` always returns a result with `offset` equal to zero and `zone` equal to `"[Unknown zone]"`, and `UTC` is `NIL`.

6.3 Tick

A `Tick.T` represents a value of a clock with subsecond resolution. The exact resolution differs from implementation to implementation and is typically one sixtieth of a second or smaller.

```
INTERFACE Tick;

IMPORT Word;

TYPE T = Word.T;

PROCEDURE Now(): T;
```

Return the current reading of the tick clock.

```
PROCEDURE ToSeconds(t: Word.T): LONGREAL;
```

Return the number of seconds in `t` *ticks.*

```
EXCEPTION Overflow;

PROCEDURE FromSeconds(s: LONGREAL): Word.T RAISES {Overflow};
```

Return the number of ticks equivalent to `s` *seconds, rounded to the nearest whole number, or raise* `Overflow` *if* `s` *is negative or the result would not be less than* `2^Word.Size`*.*

```
END Tick.
```

If `t0` is a reading of the tick clock and `t1` is another reading taken less than $2^{\texttt{Word.Size}}$ ticks after `t0`, then the number of ticks between `t0` and `t1` is `Word.Minus(t1, t0)`.

The values returned by `Tick.Now()` and `Time.Now()` typically won't stay synchronized for long periods of time. The purpose of `Tick.Now()` is to provide accurate measurements of short intervals. The purpose of `Time.Now()` is to provide "wall clock" time, preferably synchronized with UTC (coordinated universal time).

6.4 OSError

`OSError.E` is an exception raised by a number of operating system interfaces such as `File`, `FS`, and `Process`.

```
INTERFACE OSError;

IMPORT AtomList;

TYPE Code = AtomList.T;

EXCEPTION E(Code);

END OSError.
```

`E(code)` is raised by a number of methods and procedures in the operating system interfaces to signal any of an open-ended class of failures.

6.5 File

A `File.T`, or *file handle*, is a source and/or sink of bytes. File handles provide an operating-system independent way to perform raw I/O. For buffered I/O, use the `FileRd` and `FileWr` interfaces instead. A file handle is created using `OpenFile` or `OpenFileReadonly` in the `FS` interface.

```
INTERFACE File;

IMPORT Atom, OSError, Time;

TYPE
  T <: Public;
  Public = OBJECT METHODS
    read(VAR (*OUT*) b: ARRAY OF Byte;
      mayBlock: BOOLEAN := TRUE): INTEGER RAISES {OSError.E};
    write(READONLY b: ARRAY OF Byte) RAISES {OSError.E};
    status(): Status RAISES {OSError.E};
    close() RAISES {OSError.E}
  END;
  Byte = BITS 8 FOR [0 .. 255];
  Status = RECORD
    type: Type;
    modificationTime: Time.T;
    size: CARDINAL
  END;
  Type = Atom.T;

END File.
```

Formally, a file handle `h` has the components:

`type(h)`	an atom, the type of file
`readable(h)`	a boolean
`writable(h)`	a boolean
`src(h)`	(a `REF` to) a sequence of bytes
`srcCur(h)`	an integer in the range `[0..len(src(h))]`
`srcEof(h)`	a boolean
`snk(h)`	(a `REF` to) a sequence of bytes
`snkCur(h)`	an integer in the range `[0..len(snk(h))]`

The `src...` components are meaningful only if `readable(h)`. The sequence `src(h)` is zero-based: `src(h)[i]` is valid for `i` from `0` to `len(src(h))-1`. For some subtypes of `File.T`, the sequence `src(h)` can grow without bound.

The `snk...` components are meaningful only if `writable(h)`. The sequence `snk(h)` is zero based: `snk(h)[i]` is valid for `i` from `0` to `len(snk(h))-1`.

For full details on the semantics of a file handle, consult the interface defining the particular subtype, for example, `Pipe.T`, `Terminal.T`, or `RegularFile.T`. In the case where no exceptions are raised, the methods of the subtypes of `File.T` obey the following specifications:

The call

```
h.read(b, mayBlock)
```

is equivalent to

```
IF NOT readable(h) OR NUMBER(b) = 0 THEN
  Cause checked runtime error
END;
IF srcCur(h) = len(src(h)) AND NOT srcEof(h) THEN
  IF NOT mayBlock THEN RETURN -1 END;
  Block until srcCur(h) # len(src(h)) OR srcEof(h)
END;
IF srcCur(h) = len(src(h)) THEN RETURN 0 END;
Choose k such that:
  1 <= k <= MIN(NUMBER(b), len(src(h))-srcCur(h));
FOR i := 0 TO k-1 DO
  b[i] := src(h)[srcCur(h)];
  INC(srcCur(h))
END;
RETURN k
```

A result of zero always means end of file. The meaning of a subsequent read after end of file has been reached is undefined for a `File.T` but may be defined for a particular subtype.

The call

```
h.write(b)
```

is equivalent to

```
IF NOT writable(h) THEN Cause checked runtime error END;
FOR i := 0 TO NUMBER(b)-1 DO
  IF snkCur(h) = len(snk(h)) THEN
    Extend snk(h) by one byte
  END;
  snk(h)[snkCur(h)] := b[i]
  INC(srcCur(h))
END;
```

The `read` and `write` methods are not alertable because it isn't possible to alert a thread blocked in a Win32 `ReadFile` or `WriteFile` system call.

The call

```
h.status()
```

returns a result whose `type` field contains `type(h)`. See the documentation for each subtype of `File.T` for more details, including the values of the `modificationTime` and `size` fields of the result, if any.

The call

```
h.close()
```

is equivalent to

```
readable(h) := FALSE;
writable(h) := FALSE
```

Additionally, it releases any subtype-specific resources used by `h`. Every file handle should be closed.

Clients should assume that file handles are unmonitored and should avoid concurrent accesses to a file handle from multiple threads. A particular subtype of `File.T` may provide a stronger specification with respect to atomicity.

6.6 Pipe

A `Pipe.T`, or pipe, is a file handle that provides access to one endpoint of a unidirectional channel that is typically used to communicate between a parent and a child process or two sibling processes. (See `Process.Create`.)

```
INTERFACE Pipe;

IMPORT File, OSError;

TYPE T <: File.T;

VAR (*CONST*) FileType: File.Type;
```

Equal to `Atom.FromText("Pipe")`.

```
PROCEDURE Open(VAR (*OUT*) hr, hw: T) RAISES {OSError.E};
```

Create a new channel allowing bytes written to `hw` *to be read from* `hr`.

```
END Pipe.
```

Like every `File.T`, a pipe `h` has the components

`type(h)`	an atom, equal to `FileType`
`readable(h)`	a boolean
`writable(h)`	a boolean

Exactly one of `readable(h)` and `writable(h)` is true (until the pipe is closed).

A pipe `h` also has the component

`channel(h)`	a channel

If there are pipes `hw` and `hr` with `channel(hw) = channel(hr)`, `writable(hw)`, and `readable(hr)`, then a process holding `hw` can send information to a process holding `hr`.

A channel `c` has the components

`seq(c)`	a sequence of bytes
`w(c)`	a non-negative integer, the index of the next byte to write
`r(c)`	a non-negative integer, the index of the next byte to read
`nw(c)`	a non-negative integer, the number of pipes writing `c`
`nr(c)`	a non-negative integer, the number of pipes reading `c`

It is possible (but not very useful) for a channel to have values of `nw(c)` or `nr(c)` other than zero or one (see `Process.Create`).

`Open` creates a channel `c` with

```
w(c) = r(c) = 0
nw(c) = nr(c) = 1
```

and two pipes `hr` and `hw` with

```
type(hr) = type(hw) = FileType
readable(hr) = writable(hw) = TRUE
writable(hr) = readable(hw) = FALSE
channel(hr) = channel(hw) = c
```

The meaning of the call

```
h.read(b, mayBlock)
```

is given by the specification of `File.T.read` together with these definitions, where `c = channel(h)`:

```
src(h)    = seq(c)
```

```
srcCur(h) = r(c)
srcEof(h) = (nw(c) = 0)
```

Note that end-of-file is not reported until after the last pipe that can write on the channel is closed; subsequent reads are legal but always report end-of-file.

The meaning of the call

```
h.write(b)
```

is given by the specification of `File.T.write` together with these definitions, where `c = channel(h)`:

```
snk(h)    = seq(c)
snkCur(h) = w(c)
```

In some implementations, a channel has a bounded buffer, so `write` may have to block. If `nr(channel(h)) = 0`, that is, no pipe can read `h`'s channel, `write` raises `OSError.E`.

The call

```
h.status(stat)
```

assigns `FileType` to `stat.type`. Its effect on `stat.modificationTime` and `stat.size` is undefined.

The call

```
h.close()
```

is equivalent to

```
IF readable(h) THEN
  DEC(nr(channel(h)))
ELSE
  DEC(nw(channel(h)))
END;
readable(h) := FALSE;
writable(h) := FALSE
```

The channel connecting a pair of pipes is necessarily monitored, since the purpose of the channel is to allow asynchronous communication via the pipes. Nevertheless, an individual pipe should be treated as unmonitored, thus avoiding the question of the unit of atomicity for reads and writes.

6.7 Terminal

A `Terminal.T`, or terminal handle, is a file handle that provides access to a duplex communication channel usually connected to a user terminal.

```
INTERFACE Terminal;
```

```
IMPORT File;

TYPE T <: File.T;

VAR (*CONST*) FileType: File.Type;
```
Equal to `Atom.FromText("Terminal")`.

```
END Terminal.
```

Like every `File.T`, a terminal handle `h` has the components

`type(h)`	an atom, equal to `FileType`
`readable(h)`	a boolean
`writable(h)`	a boolean

A terminal handle is readable, or writable, or both (until it is closed). If it is readable, it has the component

`srcTerm(h)`	a terminal device

If it is writable, it has the component

`snkTerm(h)`	a terminal device

A terminal device `t` has the components

`seq(t)`	a sequence of bytes
`r(t)`	a non-negative integer, the index of the next byte to read
`w(t)`	a non-negative integer, the index of the next byte to write
`flag(t)`	a byte reserved to mark the end-of-file in `seq(t)`

The meaning of the call

```
h.read(b, mayBlock)
```

is given by the specification of `File.T.read` together with these definitions, where `t = srcTerm(h)`, and `k` is the number of occurrences of `flag(t)` in `seq(t)` up to `r(t)-1`:

`src(h)`	= subsequence of `seq(t)` with no occurrences of `flag(t)`
`srcCur(h)`	= `r(t)-k`
`srcEof(h)`	= `(seq(t)(r(t)) = flag(t))`

When end-of-file is reported, `r(t)` is also incremented. This means subsequent reads can return further data in `seq(t)`.

The meaning of the call

```
h.write(b)
```

is given by the specification of `File.T.write` together with these definitions, where `t = snkTerm(h)`:

```
snk(h)    = seq(t)
```

```
snkCur(h) = w(t)
```

A specific implementation may provide one or more subtypes of `Terminal.T` with additional methods.

The communication channel underlying a terminal handle is necessarily monitored, since the purpose of the channel is to allow asynchronous communication between a program and a user operating a terminal device. However a terminal handle itself should be treated as unmonitored, thus avoiding the question of the unit of atomicity for reads and writes.

6.8 RegularFile

A `RegularFile.T`, or regular file handle, provides access to a persistent extensible sequence of bytes.

```
INTERFACE RegularFile;

IMPORT File, OSError;

TYPE
  T <: Public;
  Public = File.T OBJECT METHODS
    seek(origin: Origin; offset: INTEGER): INTEGER
      RAISES {OSError.E};
    flush() RAISES {OSError.E};
    lock(): BOOLEAN RAISES {OSError.E};
    unlock() RAISES {OSError.E}
  END;
  Origin = {Beginning, Current, End};

VAR (*CONST*) FileType: File.Type;
```

Equal to `Atom.FromText("RegularFile")`.

```
END RegularFile.
```

Like every `File.T`, a regular file handle `h` has the components

`type(h)`	an atom, equal to `FileType`
`readable(h)`	a boolean
`writable(h)`	a boolean

A regular file handle `h` also has the components

`cur(h)`	an integer, the index of the next byte to read or write
`file(h)`	the identity of a regular file

There may be distinct regular file handles `h1` and `h2` with `file(h1)` equal to `file(h2)`, and more than one process may hold a single regular file handle (see `Process.Create`).

A regular file (not a handle) `f` has the components

`buffer(f)`	an extensible byte sequence
`stable(f)`	an extensible byte sequence
`mtime(f)`	a `Time.T`, the last modification time
`locked(f)`	a `Process.ID`

The sequences `buffer(f)` and `stable(f)` are zero-based and always have the same length. `stable(f)` represents the contents of the file on the disk or other persistent storage medium, while `buffer(f)` represents write-behind caching performed by the operating system. From time to time, a daemon performs

```
WITH i = some integer i in the range [0..len(buffer(f))-1] DO
  stable(f)(i) := buffer(f)(i)
END
```

The methods described in this interface are atomic with respect to the daemon.

The meaning of the call

```
h.read(b, mayBlock)
```

is given by the specification of `File.T.read` together with these definitions, where `f = file(h)`:

```
src(h)    = buffer(f)
srcCur(h) = cur(h)
srcEof(h) = TRUE
```

Because `srcEof(h)` is always `TRUE`, `read` never blocks. However, a subsequent read can return more data if an interleaved write extends `buffer(f)`. If `cur(h)` is negative (because of a prior seek), `read` raises `OSError.E`.

The meaning of the call

```
h.write(b)
```

is given by the specification of `File.T.write` together with these definitions, where `f = file(h)`:

```
snk(h)    = buffer(f)
snkCur(h) = cur(h)
```

In addition, `write` sets `mtime(file(h))` to the current time. If `write` is called when `cur(h) > size(f)` (because of a prior seek), it extends `f` with bytes of undefined value. If `cur(h)` is negative, `write` raises `OSError.E`.

The call

```
h.status(stat)
```

is equivalent to the following, in which `stat` is a local variable of type `Status`:

```
stat.type := FileType;
```

```
stat.modificationTime := mtime(file(h));
stat.size := len(buffer(file(h)));
RETURN stat
```

The call

```
h.seek(origin, offset)
```

is equivalent to

```
CASE origin OF
  Origin.Beginning => cur(h) := offset
| Origin.Current => cur(h) := cur(h)+offset
| Origin.End => cur(h) := len(buffer(file(h)))+offset
END;
RETURN cur(h)
```

Note that `seek` never changes the length of the file, although a subsequent write may do so. Use the call `h.seek(Origin.Current, 0)` to determine `cur(h)` without changing it.

The call

```
h.flush()
```

is equivalent to

```
WITH f = file(h) DO
  FOR i := 0 TO len(buffer(f))-1 DO
    stable(f)(i) := buffer(f)(i)
  END
END
```

The call

```
h.close()
```

extends the normal action of the `close` method with

```
IF locked(file(h) = Process.GetMyID() THEN
  locked(file(h)) := Process.NullID
END
```

The call

```
h.lock()
```

is equivalent to:

```
IF locked(file(h)) = Process.NullID THEN
  locked(file(h)) := Process.GetMyID();
  RETURN TRUE
END;
```

```
RETURN FALSE
```

The call

```
h.unlock()
```

is equivalent to:

```
IF locked(file(h)) # Process.GetMyID() THEN
  RAISE OSError.E
END;
locked(file(h)) := Process.NullID
```

Some implementations raise an exception if a process tries to read or write a file locked by another process. You should treat this as a checked runtime error rather than writing code to catch and recover from the exception; the same applies to unlocking a file that you didn't lock.

You lock a file with code like

```
CONST
  MaxTry = 3;
  RetryInterval = 5.0D0;
VAR try := 1;
BEGIN
  WHILE NOT h.lock() DO
    IF try=MaxTry THEN Give up END;
    INC(try);
    Time.Pause(RetryInterval)
  END;
  TRY Read or write h FINALLY h.unlock() END
END
```

The regular file underlying a regular file handle is monitored, thus allowing concurrent operations. We leave unspecified the unit of atomicity for reads and writes, so a set of processes sharing a file that needs to be updated should use the `lock` and `unlock` methods. A regular file handle itself should be treated as unmonitored. A client thread typically needs to perform a `seek` followed by a `read` or `write` as an atomic unit, which can be implemented with a mutex in the client.

6.9 Pathname

`Pathname` defines procedures for manipulating pathnames in a portable fashion.

```
INTERFACE Pathname;

IMPORT TextSeq;
```

```
TYPE
  T = TEXT;
  Arcs = TextSeq.T;
```

Most operating systems include a file system providing persistent storage (files) and naming (directories). The name space is usually a directed, rooted graph in which interior nodes are directories and exterior nodes are files and empty directories. Each arc is labeled with a character string called an arc name; the arc names in any one directory are distinct. A `Pathname.T` (or just a pathname) is a text conforming to the syntax of the underlying operating system. It consists of a sequence of arc names specifying a path starting from some distinguished directory and ending at the referent of the pathname.

A pathname may be absolute, in which case it begins with the name of a root directory. If a pathname is not absolute, it is interpreted relative to the working directory associated with the process (see `GetWorkingDirectory` in the `Process` interface).

Not all operating systems use the same syntax for pathnames, so we define the type `Arcs` to represent a pathname in a standard form allowing manipulations by portable programs. Suppose `a` is of type `Arcs`. Then `a` is non-`NIL`, `a.getlo()` indicates whether or not the pathname is absolute, and `TextSeq.Sub(a, 1)` represents a sequence (possibly empty) of arc names (all non-`NIL`). If `a` represents an absolute pathname, then `a.getlo()` is the root directory name and is non-`NIL`; if `a` represents a relative pathname, then `a.getlo()` is `NIL`.

It is often useful to view an arc name as having two parts, a base and an extension, separated by a period, for example `Pathname.i3`.

See the end of this interface for operating-system specific details.

```
EXCEPTION Invalid;
```

```
PROCEDURE Valid(pn: T): BOOLEAN;
```

Return `TRUE` iff `pn` conforms to the pathname syntax of this operating system.

When a pathname with invalid syntax is passed to a procedure in this interface not declared as raising the exception `Invalid`, the result is undefined, but safe.

```
PROCEDURE Decompose(pn: T): Arcs RAISES {Invalid};
```

Parse `pn`, returning a sequence whose first element is a root directory name (possibly `NIL`) and whose remaining elements consist of zero or more arc names. Raise `Invalid` if `Valid(pn)` is `FALSE`.

`Decompose` returns exactly the sequence of arc names present in `pn`; it doesn't attempt to produce a canonical form. Some operating systems allow zero-length arc names (see the discussion of specific systems at the end of this section.)

```
PROCEDURE Compose(a: Arcs): T RAISES {Invalid};
```

Combine the elements of `a` *to form a pathname corresponding to the syntax of this operating system. Raise* `Invalid` *if* `a` *is* `NIL`*, if* `a.getlo()` *is neither* `NIL` *nor a valid root directory name, or if one of the elments of* `TextSeq.Sub(a, 1)` *is not a valid arc name.*

```
PROCEDURE Absolute(pn: T): BOOLEAN;
```

Return `TRUE` *iff* `pn` *is an absolute pathname. Equivalent to* `Decompose(pn).getlo() # NIL`*, but faster.*

```
PROCEDURE Prefix(pn: T): T;
```

Return a pathname equal to `pn` *up to, but not including, the final arc name. If* `pn` *consists only of a root directory name,* `Prefix(pn)` *returns* `pn`*.*

```
PROCEDURE Last(pn: T): T;
```

Return the final arc name in `pn`*. If* `pn` *consists only of a root directory name,* `Last(pn)` *returns the empty string.*

```
PROCEDURE Base(pn: T): T;
```

Return a pathname equal to `pn` *except with* `Last(pn)` *replaced by its base.*

```
PROCEDURE Join(pn, base: T; ext: TEXT): T;
```

Return a pathname formed by prepending `pn` *to* `base` *(if* `pn` *is not* `NIL`*) and appending* `ext` *to* `base` *(if* `ext` *is not* `NIL`*). More precisely, this is equivalent to the following, in which* `a` *is a local variable of type* `Arcs`*:*

```
IF pn = NIL THEN a := NIL
ELSE
  IF Absolute(base) THEN Cause checked runtime error END;
  a := Decompose(pn)
END;
IF ext # NIL THEN base := base & "." & ext END;
RETURN Compose(
  TextSeq.Cat(a, TextSeq.Sub(Decompose(base), 1)))
```

The value returned by `Join` will be a valid pathname only if the `base` and `ext` conform to the syntax of the particular operating system, as specified at the end of this section.

```
PROCEDURE LastBase(pn: T): T;
```

Return the base of the final arc name of `pn`*. It is a checked runtime error if* `pn` *is empty or consists only of a root directory name.*

```
PROCEDURE LastExt(pn: T): TEXT;
```

Return the extension of the last arc name of `pn`*. It is a checked runtime error if* `pn` *is empty or consists only of a root directory name.*

```
PROCEDURE ReplaceExt(pn: T; ext: TEXT): T;
```

Return a pathname equal to `pn` *except with the extension of the final arc name replaced with* `ext`*, which must be non-*`NIL`*.*

```
VAR (*CONST*)
  Parent: TEXT;
```

A special arc name that, when encountered during a pathname lookup, stands for the parent of the directory currently being examined.

```
  Current: TEXT;
```

A special arc name that, when encountered during a pathname lookup, stands for the directory currently being examined.

```
END Pathname.
```

POSIX. Pathnames have the syntax:

```
Pathname = Absolute | Relative.
Absolute = "/" Relative.
Relative = [ArcName {"/" ArcName}].
```

`Parent` is ".." and `Current` is ".".

There is only one root directory and it is named "/". A POSIX-compliant system must support arc names at least as long as fourteen characters. An arc name longer than the maximum supported is either silently truncated by the operating system or is reported as an error, depending on a configuration option. A zero-length arc name is treated the same as ".". An arc name may contain any character except "/" and the null character, but for maximum portability the POSIX specification recommends they be restricted to upper and lower case letters, digits, and these special characters:

```
. _ -
```

Furthermore, it is recommended that arc names not start with hyphen (-).

The extension of an arc name is the suffix starting after the last ".".

The base of an arc name is the prefix up to, but not including, the final "." if the extension is nonempty; it is the entire arc name if the extension is empty.

Win32. Pathnames have the syntax, where backslash is not an escape character but a literal character:

```
Pathname = Absolute | Relative.
Absolute = Volume "\" Relative.
Relative = [ArcName {"\" ArcName}].
ArcName  = Base "." Extension | "." | "..".
Volume   = Drive ":" | "\\" Server "\" Share.
Server   = ?
Share    = ?
```

`Parent` is ".." and `Current` is ".".

The FAT (MS-DOS) file system restricts `Drive` to a single letter, and `Base` to between one and eight letters, digits, or these special characters:

```
$ % ' - _ @ { } ~ ` ! # ( )
```

`Extension` is one to three characters from the same set. Certain `Base`s, including AUX, CLOCK$, COM1, CON, LPT1, NUL, and PRN are reserved—they name devices, regardless of the directory or extension. Embedded (but not trailing) spaces are allowed in the `Base` of a file name (but not a directory name).

The HPFS and NTFS file systems allow arc names up to 254 characters, and these additional special characters are allowed:

```
, + = [ ] ;
```

Additionally, blank is significant anywhere in an arc name except at the end. Win32 allows a programmer to use either ANSI or Unicode representation for pathname strings. The NTFS file system stores full Unicode pathnames in the directories.

Macintosh. Pathnames have the syntax:

```
Pathname   = Absolute | Relative.
Absolute   = Volume ":" [ArcName {Colons ArcName}].
Relative   = ArcName
           | Colons ArcName {Colons ArcName}.
Colons     = ":" {":"}.
```

`Parent` is "::" and `Current` is ":".

A `Volume` is one to twenty-seven printing characters excluding colon (:). An arc name is one to thirty-one printing characters excluding colon. A single colon is a separator; `n+1` adjacent colons means the `n`th parent.

The extension of an arc name is the suffix starting after the last "."; if there is no ".", the extension is empty.

The base of an arc name is the prefix up to, but not including, the final "." if the extension is nonempty; it is the entire arc name if the extension is empty.

6.10 FS

The `FS` interface provides persistent storage (files) and naming (directories).

```
INTERFACE FS;

IMPORT OSError, File, Pathname, Time;

PROCEDURE GetAbsolutePathname(p: Pathname.T): Pathname.T
   RAISES {OSError.E};
```

Return an absolute pathname referring to the same file or directory as `p`.

The new pathname will not involve any symbolic links or relative arcs (that is, occurrences of `Pathname.Parent` or `Pathname.Current`.

The procedures `OpenFile` and `OpenFileReadonly` look up a pathname and return a file handle, which is an object allowing a file to be read and perhaps written. The returned value will be of some subtype of `File.T`, depending on the kind of object named by `p`. If the object is a regular file, the type will be `RegularFile.T`. If the object is a terminal, the type will be `Terminal.T`. Other, system-specific subtypes are also possible. Under appropriate conditions, `OpenFile` can create a new regular file. `OSError.E` is raised if the pathname passed to `OpenFile` or `OpenFileReadonly` is that of a directory.

```
TYPE
  CreateOption = {Never, Ok, Always};
  AccessOption = {OnlyOwnerCanRead, ReadOnly, Default};

PROCEDURE OpenFile(
    p: Pathname.T;
    truncate: BOOLEAN := TRUE;
    create: CreateOption := CreateOption.Ok;
    template: File.T := NIL;
    access: AccessOption := AccessOption.Default): File.T
  RAISES {OSError.E};
```

Return an object permitting writing and reading an existing or newly-created file named `p`.

Suppose `p` names an existing regular file. If `create = Always`, then `OSError.E` is raised. Otherwise, the existing file is opened, after truncating it to zero size if `truncate = TRUE`.

On the other hand, suppose the file named by `p` does not exist. If `create = Never`, then `OSError.E` is raised. Otherwise, a new file is created. Normally the new file is a regular file, but some implementations may determine the type of the new file from the identity of the directory in which it is being created. The access control settings of the new file are set using the values of `template` and `access`. If `template # NIL`, then `access` is ignored and the new file is given the same per-file access control settings as `template`. If `template = NIL`, the file's access control settings are determined by an implementation-defined default value, with possible restrictions determined by the value of `access`:

`OnlyOwnerCanRead` read access is allowed only by this user

`ReadOnly` write access is allowed to no one (except via the `File.T` returned by this call of `OpenFile`)

`Default` the default applies with no restrictions.

A newly-created file `f` has

```
buffer(f) = stable(f) = empty sequence
mtime(f) = current time
locked(f) = Process.NullID
```

`OpenFile` doesn't change `mtime(f)` of an existing file `f`.

If `OpenFile` returns a regular file handle, say `h`, then its initial state will be:

```
type(h) = RegularFile.FileType
readable(h) = writable(h) = TRUE
cur(h) = 0
file(h) = file with pathname p
```

To append to an existing file, perform the call

```
EVAL h.seek(Origin.End, 0)
```

after opening `h`.

```
PROCEDURE OpenFileReadonly(p: Pathname.T): File.T
  RAISES {OSError.E};
```

Return an object permitting reading the file named by `p`.

If `p` names a regular file, the call `OpenFileReadonly(p)` returns a file handle `h` with

```
type(h) = Atom.FromText("RegularFile")
readable(h) = TRUE
writable(h) = FALSE
cur(h) = 0
file(h) = file with pathname p
```

```
PROCEDURE CreateDirectory(p: Pathname.T) RAISES {OSError.E};
```

Create a directory named by `p`.

```
PROCEDURE DeleteDirectory(p: Pathname.T) RAISES {OSError.E};
```

Delete the directory named by `p`. `OSError.E` *is raised if the directory contains entries (other than perhaps* `Pathname.Current` *and* `Pathname.Parent`*).*

```
PROCEDURE DeleteFile(p: Pathname.T)
  RAISES {OSError.E};
```

Delete the file or device named by `p`. `OSError.E` *is raised if* `p` *names a directory.*

Note: Under Win32, `DeleteFile` raises `OSError.E` if `p` is open. Under POSIX, an open file may be deleted; the file doesn't actually disappear until every link (pathname) for it is deleted.

```
PROCEDURE Rename(p0, p1: Pathname.T)
  RAISES {OSError.E};
```

Rename the file or directory named `p0` *as* `p1`.

Some implementations automatically delete an existing file named `p1`, others raise `OSError.E`. Some implementations disallow a rename where `p0` and `p1` name different physical storage devices (different root directories or file systems).

```
TYPE
  Iterator <: PublicIterator;
  PublicIterator = OBJECT METHODS
    next(VAR (*OUT*) name: TEXT): BOOLEAN;
    nextWithStatus(VAR (*OUT*) name: TEXT;
      VAR (*OUT*) stat: File.Status): BOOLEAN;
    close();
  END;

VAR (*CONST*) DirectoryFileType: File.Type;
```

Equal to `Atom.FromText("Directory")`.

```
PROCEDURE Iterate(p: Pathname.T): Iterator
  RAISES {OSError.E};
```

Return an iterator for the entries of the directory named by `p`.

An `Iterator` supplies information about the entries in a directory: names and, optionally, status. The iteration does not include entries corresponding to `Pathname.Current` or `Pathname.Parent`.

The methods have the following specifications:

If more entries remain, the call `i.next(n)` sets `n` to the name of the next one and returns `TRUE`. It returns `FALSE` without setting `n` if no more entries remain.

If more entries remain, the call `i.nextWithStatus(n, s)` sets `n` to the name of the next one, sets `s` to the status of that entry, and returns `TRUE`. The value of `s.type` is `DirectoryFileType` if the entry is a directory. The call returns `FALSE` without setting `n` or `s` if no more entries remain.

The call `i.close()` releases the resources used by `i`, after which time it is a checked runtime error to use `i`. Every iterator should be closed.

You iterate over the entries in a directory with code like this:

```
VAR
  i := FS.Iterate(pathname);
  name: TEXT;
BEGIN
  TRY
    WHILE i.next(name) DO
      Process name
    END
  FINALLY
    i.close()
  END
END
```

Use `nextWithStatus` instead of `next` if you would otherwise call `Status` (or the `File.T status` method) on most of the entries (in some implementations, `nextWithStatus` requires an extra disk access).

What can be assumed if a directory is being updated concurrently with an iteration? An entry that is not inserted or deleted will occur in the iteration at least once, and an entry that occurs in the iteration must have been in the directory at some moment.

```
PROCEDURE Status(p: Pathname.T): File.Status
  RAISES {OSError.E};
```

Return information about the file or directory named by `p`.

Possible values of `stat.type` include

```
FS.DirectoryFileType (a directory)
RegularFile.FileType (a disk file)
Terminal.FileType (a terminal)
```

If `p` is a disk file, `stat.modificationTime` and `stat.size` will be set.

See also the `status` method of `File.T` and the `nextWithStatus` method of `Iterator`.

```
PROCEDURE SetModificationTime(
    p: Pathname.T;
    READONLY t: Time.T)
  RAISES {OSError.E};
```

Change the modification time of the file or directory named by `p` *to* `t`.

```
END FS.
```

6.11 Process

A process is the execution of a program by one or more threads within an address space. A process may hold a variety of resources such as file handles.

```
INTERFACE Process;

IMPORT File, OSError, Pathname;

TYPE T <: REFANY;
```

A `Process.T`, or process handle, provides access to a child process.

```
PROCEDURE Create(
    cmd: Pathname.T;
    READONLY params: ARRAY OF TEXT;
    env: REF ARRAY OF TEXT := NIL;
    wd: Pathname.T := NIL;
    stdin, stdout, stderr: File.T := NIL): T
  RAISES {OSError.E};
```

Create a new process and cause it to execute the program with pathname `cmd`, parameters `params`, environment variables `env`, working directory `wd`, and standard file handles `stdin`, `stdout`, and `stderr`. Return the handle of the new process.

If `cmd` consists of a single (relative) arc name, then it is looked up in an operating-system dependent way (see below). Otherwise, `cmd` is looked up in the normal fashion as an absolute pathname or as a pathname relative to the current working directory (not `wd`).

A process can examine its own parameters via the interface `Params`. The parameter `params[i]` passed to `Create` will correspond to the value of `Params.Get(i+1)` in the newly created process (because `Params.Get(0)` returns the command name). (See the `Params` interface for the way SRC Modula-3 treats parameters beginning with the characters `@M3`.)

If `env` is not `NIL`, it consists of a reference to an array of texts that must have the form `name=value`. If `env` is `NIL`, it defaults to the environment variables of the caller's process. A process can examine its own environment variables via the interface `Env`.

If `wd` is `NIL`, it defaults to the working directory of the caller's process.

If any of `stdin`, `stdout`, or `stderr` are `NIL`, the corresponding file handle of the new process is `NIL`. A process can obtain its own standard file handles by calling the procedure `GetStandardFileHandles` defined later in this interface.

The sharing established by passing a `File.T` to a new process requires care. For example, seeks done by either process affect both, and passing a `Pipe.T` increments a reference count of the underlying channel. See the end of this interface for an example of using `Create` with pipes.

POSIX. `Create` forks a child process, which executes the specified command. If `cmd` consists of a single (relative) arc name, `Create` searches each of the directories specified by the PATH environment variable for a file named `cmd` that is executable by the current (effective) user. If the attempt to execute the command returns the Unix error ENOEXEC, then the child process executes `/bin/sh` with the original arguments prefixed by the pathname determined earlier.

Win32. `Create` calls `Win32.CreateProcess`. If `cmd` consists of a single (relative) arc name, `Win32.CreateProcess` first appends `.EXE` if `cmd` includes neither an extension nor a final period, and then searches for this name in the following sequence of directories: the working directory; the Windows system directory; the Windows directory; the directories listed in the PATH environment variable.

```
TYPE ExitCode = [0 .. 16_7FFFFFFF];
```

An exit code (or status) of zero normally means successful termination, and a non-zero value normally indicates an error, but the exact conventions vary between systems and programs.

```
PROCEDURE Wait(p: T): ExitCode;
```

Wait until the process with handle `p` *terminates, then free the operating system resources associated with the process and return an exit code indicating the reason for its termination. It is a checked runtime error to call* `Wait` *twice on the same process handle.*

POSIX. The value returned by `Wait` is equal to the `status` result of the `wait` system call.

Win32. The value returned by `Wait` is `c MOD (LAST(ExitCode) + 1)` where `c` is the value returned by `Win32.GetExitCodeProcess`.

```
PROCEDURE Exit(n: ExitCode := 0);
```

Call the registered exitors and terminate the program with exit code `n`*. Terminating a Modula-3 program by "falling off the end" is equivalent to calling* `Exit(0)`*.*

```
PROCEDURE Crash(msg: TEXT);
```

Call the registered exitors and terminate the program with the error message `msg`*. If possible, invoke a debugger or generate a core dump.*

Modula-3 implementations that don't convert checked runtime errors into exceptions should call `Crash` to abort the program.

Some Modula-3 implementations catch external events (e.g. Unix signals) or internal interrupts (e.g. floating-point underflow) and call `Crash`. Consult your local installation guide for more information.

```
PROCEDURE RegisterExitor(p: PROCEDURE());
```

Register the procedure `p` *to be called when* `Exit` *or* `Crash` *is called.*

Each registered exitor is called at most once. Exitors are called in reverse of the order they were registered. A facility implementing a class of objects should register only a single exitor, which can consult a private data structure to determine which of its objects need cleanup. `RegisterExitor` should be called at module initialization time (not when the first object is created) to guarantee the correct registration order.

```
TYPE ID = [0 .. 16_7FFFFFFF];
CONST NullID: ID = 0;
```

An `ID` or process identifier is assigned to each process when it is created. At any moment, no two processes on the same computer have the same identifier, but identifiers can be reused over time. No process is ever assigned the identifier `NullID`.

```
PROCEDURE GetID(p: T): ID;
```

Return the process identifier of the process with handle `p`*.*

```
PROCEDURE GetMyID(): ID;
```

Return the process identifier of the caller's process.

```
PROCEDURE GetStandardFileHandles(
    VAR (*OUT*) stdin, stdout, stderr: File.T);
```

Return the standard input/output handles that were supplied when this process was created.

```
PROCEDURE GetWorkingDirectory(): Pathname.T
```

```
  RAISES {OSError.E};
```

Return an absolute pathname for the working directory of the caller's process.

```
PROCEDURE SetWorkingDirectory(path: Pathname.T)
  RAISES {OSError.E};
```

Change the working directory of this process to `path`.

```
END Process.
```

Example. A typical use of `Create` is to run a filter process that reads from standard input and writes a transformed version to standard output. The first step is to create two sets of pipes to carry the standard input and standard output of the new process. (If desired, standard error can be handled in the same way as standard output.)

```
VAR hrChild, hwChild, hrSelf, hwSelf: Pipe.T;
BEGIN
  Pipe.Open(hr := hrChild, hw := hwSelf);
  Pipe.Open(hr := hrSelf, hw := hwChild);
```

The next step is to create the process, passing the appropriate pipes, and then to close the original instances of these pipes. (The pipes must be closed to maintain the correct reference counts on the underlying channels.)

```
  WITH p = Process.Create(..., hrChild, hwChild, NIL) DO
    TRY
      TRY hrChild.close(); hwChild.close()
      EXCEPT OSError.E => (*SKIP*)
      END;
```

Now comes the actual writing and reading, which is conveniently performed using I/O streams:

```
      WITH wr = NEW(FileWr.T).init(hwSelf),
           rd = NEW(FileRd.T).init(hrSelf) DO
        Write wr (and perhaps read rd)
```

Closing `wr` causes the filter to encounter end-of-file on its standard input, which should cause it to flush its standard output and terminate. This in turn causes this process to read end-of-file.

```
        TRY Wr.Close(wr)
        EXCEPT Wr.Failure, Thread.Alerted => (*SKIP*)
        END;
```

```
              Read rd to end-of-file;
              TRY Rd.Close(rd)
              EXCEPT Rd.Failure, Thread.Alerted => (*SKIP*)
              END
            END
```

The last step is to clean up the process.

```
          FINALLY EVAL Process.Wait(p)
          END
        END
      END
```

6.12 Params

This interface provides access to the command line arguments given to a process when it is started (see `Process.Create`).

```
INTERFACE Params;

VAR (*CONST*) Count: CARDINAL;
```

Parameters are indexed from `0` (the command name) to `Count-1`.

```
PROCEDURE Get(n: CARDINAL): TEXT;
```

Return the parameter with index `n`. It is a checked runtime error if `n >= Count`.

```
END Params.
```

Parameters that begin with the characters `@M3` are reserved for use by the SRC Modula-3 runtime. They are not included in the value of `Count` or in the sequence indexed by `Get`.

6.13 Env

This interface provides access to the environment variables given to a process when it is started (see `Process.Create`).

```
INTERFACE Env;

PROCEDURE Get(nm: TEXT): TEXT;
```

Return the value of the environment variable whose name is equal to `nm`, or `NIL` if there is no such variable.

```
VAR (*CONST*) Count: CARDINAL;
```

Environment variables are indexed from `0` *to* `Count-1`*.*

```
PROCEDURE GetNth(n: CARDINAL; VAR (*OUT*) nm, val: TEXT);
```

Set `nm` *and* `val` *to the name and value of the environment variable with index* `n`*. It is a checked runtime error if* `n >= Count`*.*

```
END Env.
```

7 Runtime

7.1 WeakRef

Most Modula-3 programs simply let the garbage collector deallocate storage automatically, but some programs need more control. For example, if a variable allocated in the traced heap contains a handle on a resource in the operating system or in some other address space, then when the variable is garbage-collected it may be important to deallocate the resource. The `WeakRef` interface provides this additional control.

A *node* is a datum allocated on the traced heap. Thus a node is either the referent of a variable of a fixed reference type or the data record of a traced object. Note that a node is not a Modula-3 reference, but the allocated storage to which a reference can refer.

A `WeakRef.T` is a data structure that refers to a node without protecting the node from the garbage collector. If `w` is a weak reference, we write `nd(w)` to denote the node to which `w` refers.

We say that a weak reference `w` *dies* at the moment that the garbage collector detects that `nd(w)` is unreachable. A precise definition of unreachable is given below. Once a weak reference has died, it remains dead forever, even if the node to which it refers becomes reachable again.

Associated with each weak reference `w` is a *cleanup procedure* `cp(w)`. If the cleanup procedure is not `NIL`, the garbage collector will schedule a call to it when the weak reference dies.

```
INTERFACE WeakRef;

TYPE T =
  RECORD
    byte: ARRAY [0..7] OF BITS 8 FOR [0..255]
  END;
```

Please treat this as though it were an opaque type: the only operations allowed are assignment, equality tests, and the procedures in this interface.

```
PROCEDURE FromRef(r: REFANY; p: CleanUpProc := NIL): T;
```

Return a weak reference `w` *such that* `nd(w) = r` *and* `cp(w) = p`*. It is a checked runtime error if* `r` *is* `NIL`*. It is illegal to create more than one weak reference with a non-nil cleanup to the same node; violations of this rule may lead to a checked runtime error, or may cause one of the cleanup actions to be omitted.* `FromRef` *is not necessarily functional: it is possible that* `nd(w1) = nd(w2)` *but* `w1 # w2`*.*

```
PROCEDURE ToRef(w: T): REFANY;
```

Return a reference to `nd(w)`*, unless* `w` *is dead, in which case return* `NIL`*.*

```
TYPE CleanUpProc = PROCEDURE(READONLY w: T; r: REFANY);
```

If `cp(w)` is not `NIL`, then when `w` dies, the garbage collector will schedule the call `cp(w)(w, <reference to nd(w)>)`.

```
END WeakRef.
```

The cleanup procedure will be executed at some point after the weak reference dies. A cleanup procedure is called with no locks held; it must return promptly to allow other objects to be cleaned up.

The computation `cp(w)(w, ref)` is allowed to store `ref` in a non-local variable, thus making `nd(w)` reachable again; the heap storage will not have been freed. This does not change the fact that `w` is dead. The cleanup procedure can re-enable cleanup, if desired, by creating a new weak reference to `nd(w)`.

The storage for a node is reclaimed when it is unreachable and all weak references to it are dead and all cleanup calls scheduled for it have been completed.

Finally we come to the precise definition of "reachable":

A node is *reachable* if it can be reached by a path of traced references starting from a current procedure activation record, a global variable, or a weakly referenced node with a non-nil cleanup *other than itself*.

Thus a weak reference to a node `nd` does not make `nd` reachable, but if it has a non-nil cleanup, it makes other nodes referenced from `nd` reachable.

For example, if `A` and `B` are two nodes that are weakly referenced by weak references with non-nil cleanup procedures, then if `B` is reachable from `A`, then `B` is reachable. But if `A` is not reachable, then the garbage collector will eventually detect this and schedule the cleanup of `A`. If the cleanup call returns without resurrecting `A`, then `A`'s storage will be reclaimed, at which point `B` will be unreachable, which will lead to its cleanup.

If `A` and `B` are weakly referenced nodes with non-nil cleanups that are connected by a cycle of traced references, then both of them are reachable. As long as the cycle persists, neither will be cleaned up. This situation represents a storage leak and should be avoided.

Examples

1. Suppose you want writers of the class `WrX.T` to be automatically flushed and closed if they become unreachable. Then you could write code like the following in the `WrX` module:

```
MODULE WrX; IMPORT WeakRef, Wr, ...;

PROCEDURE New(...): T =
  VAR res := NEW(T); BEGIN
    (* ... initialize res as a WrX.T ... *)
    EVAL WeakRef.FromRef(res, Cleanup);
```

```
    RETURN res
  END New;

PROCEDURE Cleanup(self: WeakRef.T; ref: REFANY) =
  VAR wr: T := ref; BEGIN
    IF NOT Wr.Closed(wr) THEN
      Wr.Flush(wr);
      Wr.Close(wr)
    END
  END Cleanup;
```

There is no danger that another thread could close the writer after the test `NOT Wr.Closed(wr)` and before the call `Wr.Flush(wr)`, since when `Cleanup` is called, the writer is unreachable. Therefore the cleanup method has exclusive access to the writer.

2. The network object runtime must map wire representations for network objects into surrogate objects. To hand out the same surrogate for the same wire representation, it keeps a table mapping wire representations to surrogates. This table contains weak references, so the table entry itself does not prevent the surrogate from being collected. When the surrogate is collected, it is removed from the table and the server containing that object is notified that the client no longer has a surrogate for it.

When a weak reference in the table becomes dead, the network object represented by the dead surrogate might be unmarshaled by the address space before the surrogate is cleaned up. In this case the unmarshaling code resurrects the unreachable surrogate by creating a new weak reference and inserting it in the table in place of the dead weak reference. The cleanup code can tell whether to report in clean by checking whether there is a new weak reference in the table or not.

Here is a sketch of the code:

```
TYPE Surrogate = OBJECT wr: WireRep; ... END;

VAR
  mu := NEW(MUTEX);
  <* LL >= {mu} *>
  tbl := NEW(WireRepToWeakRefTbl.T);
```

The mutex `mu` must be held to read or write `tbl` (that is what the `LL` pragma means).

The table `tbl` maps `WireRep`s to `WeakRef`s that reference surrogates.

The following invariants hold whenever `mu` is not held:

If `tbl(wrep)` is not dead, then `nd(tbl(wrep))` is the surrogate for the network object whose wire representation is `wrep`.

If `tbl(wrep)` is dead, then the surrogate for `wrep` is unreachable.

If `tbl` has no entry for `wrep`, then the address space contains no surrogate for `wrep`.

```
PROCEDURE Cleanup(wref: WeakRef.T; ref: REFANY) =
<* LL = {} *>
  VAR
    srg := NARROW(ref, Surrogate);
    tblVal: WeakRef.T;
  BEGIN
    LOCK mu DO
      IF tbl.get(srg.wr, tblVal) AND wref = tblVal
      THEN
        EVAL tbl.delete(srg.wr);
        ... Report that srg is deleted ...
      END
    END
  END Cleanup;

PROCEDURE WireRepToSrg(wrep: WireRep): Surrogate =
  VAR wref: WeakRef.T; res: Surrogate; BEGIN
    LOCK mu DO
      IF tbl.get(wrep, wref) THEN
        res := WeakRef.ToRef(wref);
        IF res # NIL THEN RETURN res END
      END;
      res := NewSurrogate(wrep);
      EVAL tbl.put(wrep, WeakRef.FromRef(res, Cleanup));
      RETURN res
    END
  END WireRepToSrg;
```

In the above we assume that `NewSurrogate` creates a new surrogate from a wire representation.

The remaining interfaces in this section provide the low-level features needed to implement pickles and network objects. Most programmers won't directly use any of these interfaces.

7.2 RTType

`RTType` provides access to the runtime type system.

Each reference type is assigned a unique typecode. A typecode is "proper" if it lies in the range `[0..MaxTypecode()]`. The proper typecodes include all those that correspond to actual types in the running Modula-3 program. Other typecodes, proper and improper, may be used internally by the runtime system and garbage collector.

Although the language requires that typecodes exist only for object types and for traced reference types (including `NULL`), the implementation of `RTType` also provides typecodes for untraced reference types.

The values returned by the builtin operation `TYPECODE` correspond to (a subset of) the proper typecodes.

```
INTERFACE RTType;

IMPORT RT0;

TYPE Typecode = RT0.Typecode;

CONST NoSuchType: Typecode = LAST(Typecode);
```

A reserved typecode that represents unknown types.

```
PROCEDURE MaxTypecode(): Typecode;
```

Return the largest proper typecode.

```
PROCEDURE IsSubtype(a, b: Typecode): BOOLEAN;
```

Return `TRUE` iff the type corresponding to `a` is a subtype of the type corresponding to `b`. It is a checked runtime error if either `a` or `b` is not a proper typecode.

```
PROCEDURE Supertype(tc: Typecode): Typecode;
```

Return the typecode of the declared supertype of the object type corresponding to `tc`. If `tc` corresponds to `ROOT`, `UNTRACED ROOT` or a non-object reference type, return `NoSuchType`. It is a checked runtime error if `tc` is not a proper typecode.

```
PROCEDURE IsTraced(tc: Typecode): BOOLEAN;
```

Return `TRUE` iff the type corresponding to `tc` is traced.

```
PROCEDURE Get(tc: Typecode): RT0.TypeDefn;
```

Return a pointer to the typecell with typecode `tc`. It is a checked runtime error to pass an improper typecode.

```
PROCEDURE GetNDimensions(tc: Typecode): CARDINAL;
```

Return the number of open dimensions of the open array type that corresponds to `tc`'s referent. If `tc`'s referent is not an open array, return `0`.

```
END RTType.
```

7.3 RTAllocator

`RTAllocator` provides access to the runtime storage allocator.

```
INTERFACE RTAllocator;
```

```
FROM RTType IMPORT Typecode;
```

Each of the procedures described below allocates and initializes heap storage. Calling any of these procedures with a typecode `tc` that names a type `T` is equivalent to calling `NEW` for that type. It is a checked runtime error to pass a typecode that is not proper. (See `RTType` for the definition of proper typecode.)

```
PROCEDURE NewTraced(tc: Typecode): REFANY;
```

Return a reference to a freshly allocated and initialized, traced referent with typecode `tc`. It is a checked runtime error if `tc` does not name a traced reference type other than `REFANY`, or if its referent is an open array.

```
PROCEDURE NewUntraced(tc: Typecode): ADDRESS;
```

Return a reference to a freshly allocated and initialized, untraced referent with typecode `tc`. It is a checked runtime error if `tc` does not name an untraced reference type other than `ADDRESS`, or if it names an untraced object type, or if its referent is an open array.

```
PROCEDURE NewUntracedObject(tc: Typecode): UNTRACED ROOT;
```

Return a freshly allocated and initialized, untraced object with typecode `tc`. It is a checked runtime error if `tc` does not name an untraced object type.

```
TYPE Shape = ARRAY OF INTEGER;
```

```
PROCEDURE NewTracedArray(
    tc: Typecode;
    READONLY s: Shape): REFANY;
```

Return a reference to a freshly allocated and initialized, traced open array referent with typecode `tc` and sizes `s[0]`, ..., `s[LAST(s)]`. It is a checked runtime error if `tc` does not name a traced reference to an open array, or if any `s[i]` is negative, or if `NUMBER(s)` does not equal the number of open dimensions of the array.

```
PROCEDURE NewUntracedArray(
    tc: Typecode;
    READONLY s: Shape): ADDRESS;
```

Return a reference to a freshly allocated and initialized, untraced open array referent with typecode `tc` and sizes `s[0]`, ..., `s[LAST(s)]`. It is a *checked runtime error if* `tc` *does not name an untraced reference to an open array, or if any* `s[i]` *is negative, or if* `NUMBER(s)` *does not equal the number of open dimensions of the array.*

```
END RTAllocator.
```

7.4 RTCollector

`RTCollector` provides control over the Modula-3 garbage collector.

```
INTERFACE RTCollector;
```

The purpose of a garbage collector is to reclaim unreachable nodes on the traced heap; most Modula-3 programs could not run very long without a collector. Even so, automatic garbage collection has some practical drawbacks.

1. The collector might move heap nodes to different addresses. This is usually unnoticable to programs, but can cause problems when programs must work with the addresses of heap nodes, since it is not guaranteed that `ADR(x^)` is a constant over the lifetime of `x^`. There are two main cases when programs must work with such addresses.
   (a) To implement hash tables, etc.
   (b) To pass addresses to procedures written in other languages, which is inherently unportable.
2. Unsafe code can put the traced heap temporarily into an inconsistent state. If the collector happens to run then, it might delete nodes that seem unreachable but that in fact are accessible. Of course, unsafe code itself is inherently unportable.

This interface allows the program to control the Modula-3 collector to avoid such problems, as well as to pass hints to improve performance.

Disabling the collector. The collector is initially enabled; the collector can reclaim storage, and move nodes in memory. While the collector is disabled, there will be no time spent in the collector. Allocation in the traced heap may proceed normally, although the heap will grow without bound. Nodes unreachable by the Modula-3 rules will not be reclaimed, and no nodes will move.

```
PROCEDURE Disable();
```

Disable the collector.

```
PROCEDURE Enable();
```

Reenable the collector if `Enable` *has been called as many times as* `Disable`*. It is a checked runtime error to call* `Enable` *more times than* `Disable`*.*

Disabling motion. Disabling motion gives fewer guarantees than disabling the collector; while motion is disabled, it is guaranteed only that no nodes will move. Disabling motion is no more expensive than disabling the entire collector, and may be cheaper in some implementations.

```
PROCEDURE DisableMotion();
```

Disable motion. While motion is disabled, no nodes will move.

```
PROCEDURE EnableMotion();
```

Reenable motion if `EnableMotion` *has been called as many times as* `DisableMotion`*, and* `Enable` *has been called as many times as* `Disable`*. It is a checked runtime error to call* `EnableMotion` *more times than* `DisableMotion`*.*

Collecting. Calling `Collect` is a hint from the program that now would be a good time for a collection (for example, if a large amount of storage has become unreachable, or if the program expects to wait some time for an external event).

```
PROCEDURE Collect();
```

Maybe collect now.

```
END RTCollector.
```

Implementation notes. This section describes the implementation of the SRC Modula-3 collector, as a guide to SRC Modula-3 programmers and as an indication of how this interface is matched to a particular implementation. Portable programs must not take advantage of implementation details of the SRC Modula-3 collector.

The SRC Modula-3 collector is an incremental, generational, conservative mostly-copying collector that uses VM protection on heap pages to be notified of certain heap accesses.

Because the SRC collector is conservative, an inaccessible node may be considered reachable if a bit-pattern either on a thread's stack or in its registers might be a reference to or into the node. Experience to date has not shown accidental node retention to be a problem.

The SRC collector will not collect or move a node while any thread's stack or registers contains a reference to or into the node. The SRC Modula-3 system guarantees that this will include references passed as value parameters. This guarantee is useful for calling foreign procedures.

`Disable` completes the current incremental collection, if any, and unprotects all heap pages, so that no page faults will occur while collection is disabled. No new collections will start while collection is disabled. The next collection after collection is reenabled will be total, as opposed to partial, since unprotecting the heap loses generational information.

`DisableMotion` disables further collections from beginning. `DisableMotion` does not finish the current incremental collection, since the collector already guarantees that the program will not see addresses in the previous space. No new collections will start while motion is disabled, so that the current space will not become the previous space. It is not necessary to unprotect the heap.

`Collect` completes the current incremental collection, if any, then performs a total collection before returning to the caller.

The `@M3nogc` flag performs an initial call to `Disable`.

The SRC collector also supports additional operations for controlling the frequency of collection, disabling and reenabling incremental and generational collection, reporting on collector performance, and so on. These operations are accessible through the implementation-dependent `RTCollectorSRC` interface.

7.5 RTHeap

`RTHeap` provides access to the layout of data on the heap.

Each referent on the heap, and the heap data record for each object, is represented as a contiguous sequence of "data bytes". Referents and data records may also contain other "non-data" bytes like headers, method suite pointers, or open array shapes.

See `RTType` for related operations on types.

```
INTERFACE RTHeap;
```

```
PROCEDURE GetDataAdr(r: REFANY): ADDRESS;
```

If `r` is a traced reference, returns the address of `r^`'s data bytes. If `r` is a traced object, returns the address of the bytes of `r`'s data record. It is a checked runtime error if `r` is `NIL`. Note that the address can subsequently change unless object mobility is disabled using `RTCollector`.

```
PROCEDURE GetDataSize(r: REFANY): CARDINAL;
```

If `r` is a traced reference, returns the number of `r^`'s data bytes. If `r` is a traced object, returns the number of bytes of `r`'s data record. It is a checked runtime error if `r` is `NIL`.

```
PROCEDURE GetArrayShape(r: REFANY; VAR s: ARRAY OF INTEGER);
```

If `r` is a traced reference to an open array, returns in `s[0 .. n-1]` the size of each dimension of the n-dimensional open array `r^`. If `s` is too large, the extra elements are ignored; if it's too small, the extra sizes are discarded. It is a checked runtime error if `r` is `NIL`. If `r` is not a reference to an open array, `s` is unchanged.

```
END RTHeap.
```

7.6 RTTypeFP

`RTTypeFP` provides runtime access to type fingerprints.

A type's fingerprint is a 64-bit checksum computed from its declaration. The probability of distinct types having the same fingerprint is very small. See the `Fingerprint` interface for more details.

Typecodes may vary between executions of a program but fingerprints do not. Fingerprints are portable across multiple runs of a single program and across all programs compiled by the same compiler.

```
INTERFACE RTTypeFP;

IMPORT Fingerprint;
FROM RTType IMPORT Typecode;

PROCEDURE ToFingerprint(tc: Typecode): Fingerprint.T;
```

Return the fingerprint corresponding to `tc`. It is a checked runtime error if `tc` is not proper or does not name a traced reference type.

```
PROCEDURE FromFingerprint(READONLY fp: Fingerprint.T)
  : Typecode;
```

Return the typecode that corresponds to `fp`. If no such typecode exists, returns `RTType.NoSuchType`.

```
END RTTypeFP.
```

A Basic Data Types

An `Integer.T` is an `INTEGER`. This interface is intended to be used to instantiate generic interfaces and modules such as `Table` and `List`.

```
INTERFACE Integer;

IMPORT Word;

TYPE T = INTEGER;

PROCEDURE Equal(a, b: T): BOOLEAN;
```
Return `a = b`.

```
PROCEDURE Hash(a: T): Word.T;
```
Return `a`.

```
PROCEDURE Compare(a, b: T): [-1..1];
```
Return -1 if `a < b`, *0 if* `a = b`, *or +1 if* `a > b`.

```
END Integer.
```

A `Refany.T` is a `REFANY`. This interface is intended to be used to instantiate generic interfaces and modules such as `Table` and `List`.

```
INTERFACE Refany;

IMPORT Word;

TYPE T = REFANY;

PROCEDURE Equal(r1, r2: T): BOOLEAN;
```
Return `r1 = r2`.

```
PROCEDURE Hash(r: T): Word.T;
```
Cause a checked runtime error.

```
PROCEDURE Compare(r1, r2: T): [-1..1];
```
Cause a checked runtime error.

```
END Refany.
```

Note that the interfaces `Text`, `Real`, `LongReal`, `Extended`, and `Atom` (which were presented in the main body of this report) provide `Equal`, `Hash`, and `Compare` procedures.

References

[1] Cecilia Aragon and Raimund Seidel. Randomized search trees. In *Proceedings 30th FOCS*, 1989.

[2] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th Symposium on Operating Principles*, December 1993.

[3] Andrei Broder. Some applications of Rabin's fingerprinting method. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[4] Marc H. Brown and James R. Meehan (editors). VBTkit reference manual. Research report, Digital Equipment Corporation Systems Research Center. To appear.

[5] Marc H. Brown and James R. Meehan. FormsVBT reference manual. Research report, Digital Equipment Corporation Systems Research Center. To appear.

[6] David M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, November 30 1990.

[7] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), October 1974.

[8] IEEE Technical Committee on Operating Systems. *Standard Portable Operating System Interface for Computer Environments.* IEEE, 1988. Standard 1003.1-1988.

[9] Donald E. Knuth. *Seminumerical Algorithms.* Addison Wesley, second edition, 1981.

[10] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2), February 1980.

[11] Mark S. Manasse and Greg Nelson. Trestle reference manual. Research Report 68, Digital Equipment Corporation Systems Research Center, December 1991.

[12] Microsoft Corporation. *Microsoft Win32 Programmer's Reference.* Microsoft Press, 1993.

[13] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[14] M. O. Rabin. Fingerprinting by random polynomials. Report TR-15-81, Department of Computer Science, Harvard University, 1981.

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

[16] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[17] Guy L. Steele Jr. and Jon L White. How to print floating-point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 1990.

[18] J. E. Stoy and C. Strachey. OS6—an experimental operating system for a small computer. Part 2: Input/output and filing system. *The Computer Journal*, 15(3), May 1972.

Index